# 1. díl - Úvod do jazyka Java

Vítejte u prvního dílu seriálu o Javě. Budeme se učit postupně, od úplných začátků až po složité konstrukce, objektové modely a např. práci s databází. S trochou trpělivosti a vytrvalosti se z tebe tak stane dobrý programátor.

Abychom plně poruzuměli jazyku Java, ohlédněme se do minulosti na to, jak se programovací jazyky vyvíjely. Bude pro nás totiž důležité pochopit, jak Java pracuje a proč je dobré programovat právě v tomto jazyce.

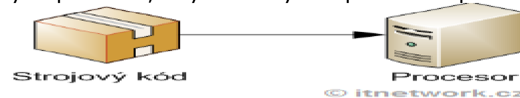
Vývoj programovacích jazyků

1. generace jazyků - Strojový kód

Procesor počítače umí vykonávat jen omezené množství jednoduchých instrukcí, které jsou uloženy jako sekvence bitů, jsou to tedy čísla. Ta se mu obvykle zadávají v hexadecimální (šestnáctkové) soustavě. Instrukce jsou tak elementární, že umožňují pouze např. sčítání adres nebo skoky mezi instrukcemi. Nelze např. jednoduše sečíst dvě čísla, musíme se na čísla dívat jako na adresy v paměti a takové sečtení čísel zabere několik instrukcí. Program sčítající dvě čísla by vypadal např. takto:

```
2104
1105
3106
7001
0053
FFFE
0000
```

Instrukce se procesoru předloží v binární podobě. Takovýto kód je samozřejmě extrémně nečitelný a závisí na instrukční sadě daného CPU. Určitě v tomto jazyce nebude jednoduché tvořit nějaké programy, bohužel **každý** program musí být nakonec do tohoto jazyka přeložen, aby mohl být na procesoru počítače spuštěn.




2. generace jazyků - Assembler

Assembler (zkráceně ASM) není o nic jednodušší, než strojový kód, ale je lidsky čitelný. Jedná se o strojový kód, ve kterém mají instrukce slovní označení (kód), čili si člověk nemusí pamatovat čísla. Kódy instrukcí se poté přeloží na výše uvedený strojový kód. Stejný program by v ASM vypadal takto:

```
ORG 100
LDA A
ADD B
STA C
HLT
DEC 83
DEC –2
DEC 0
END
```

Vidíme, že je to poněkud lidštější, ale stále nezasvěcení lidé vůbec netuší, jak program funguje (včetně mne).

3. generace jazyků

Jazyky v třetí generaci konečne nabízí uživateli určitou abstrakci nad tím, jak program vidí počítač, zaměřují se na to, jak program vidí člověk. Naše čísla jsou vnímána již jako proměnné, zdrojový kód připomíná matematický zápis.

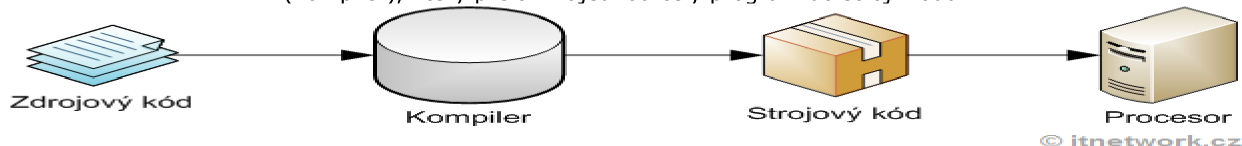
Sečtení dvou čísel by v jazyce C vypadalo takto:

```c
int main(void)
{
int a, b, c;
a = 83;
b = -2;
c = a + b;
return 0;
}
```

Všichni asi tušíme, co program dělá, sečte čísla 83 a -2 a výsledek uloží do proměnné c. U všech jazyků třetí generace je samozřejmě výhodou vysoká čitelnost. S dalším vývojem šly jazyky ještě dál a přinesly objektově orientované programování, ale o tom až později. Jazyky v třetí generaci spadají v zásadě do třech kategorií:

Kompilované jazyky

Kompilované (neřízené) jazyky mají tedy svůj zdrojový kód v jazyce, kterému lidé dobře rozumí. Tento zdrojový kód se samozřejmě musí přeložit do strojového kódu, aby ho bylo možné na procesoru spustit. Tento překlad zajišťuje překladač (kompiler), který přeloží najednou celý program do stroj. kódu.




Kompilace má tyto **výhody**:

- **Rychlost** - Jediné zbrždění spočívá v jednorázové kompilaci, přeložený program poté běží srovnatelně rychle, jako kdyby byl napsán např. v ASM.
- **Nepřístupnost zdroj. kódu** - Program se šíří již zkompilovaný, není jej možné jednoduše modifikovat pokud zároveň nevlastníte jeho zdroj. kód.
- **Snadné odhalení chyb ve zdroj. kódu** - Pokud zdrojový kód obsahuje chybu, celý proces kompilace spadne a programátor je s chybou seznámen. To značně zjednodušuje vývoj.

Dále jsou tu samozřejmě **nevýhody**:

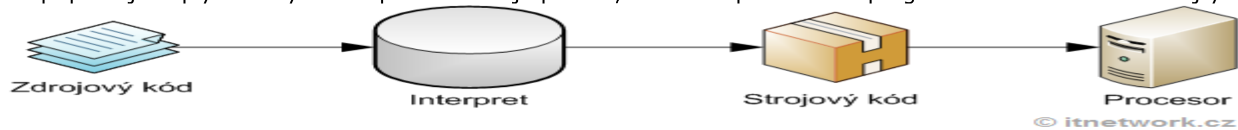
- **Závislost na platformě** - Program je stále závislý na platformě, tedy na typu procesoru a operačním systému. Zkompilovaný program nemůžeme vzít a přenést na jinou platformu bez toho, aby byl na této platformě zkompilován.
- **Nemožnost editace** - Jakmile se program jednou zkompiluje do strojového kódu, nelze ho editovat jinak, než opětovnou kompilací. To pochopitelně platí i pro výše zmíněné jazyky.

- **Memory management** - Vzhledem k tomu, že počítač danému programu nerozumí a jen mechanicky vykonává instrukce, můžeme se někdy setkat s velmi nepříjemnými chybami s přetečením paměti. Kompilované jazyky obvykle nemají automatickou správu paměti a jsou to jazyky nižší (s nižším komfortem pro programátora). Běhové chyby způsobené zejména špatnou správou paměti se kompilací neodhalí.

Příkladem kompilovaných jazyků jsou např. jazyk C, jeho objektový následník C++ nebo Pascal/Delphi.

Interpretované jazyky

Interpretace se snaží řešit problém přenositelnosti programů mezi různými platformami a také přichází s vyšším komfortem pro programátora. Interpret funguje podobně, jako kompiler, jen nepřekládá program celý najednou, ale překládá pouze to, co je v danou chvíli potřeba. (Interpreter znamená v angličtině tlumočník, tedy nejprve vyslechne jednu větu mluvčího a tu poté přeloží a vysloví. Překlad probíhá během proslovu, tedy běhu programu, po větách/instrukcích. Kompiler/překladač přeloží rozhovor celý najednou a poté ho celý přečte.). Můžeme si představit, že výše uvedený zdrojový kód by interpret četl po jednotlivých řádcích, tu část by vždy zkompiloval do strojového kódu a vykonal. Výsledek kompilace by zahodil a přesunul by se na další řádek. Možná vám to připadá jako plýtvání výkonem procesoru a je pravda, že tento způsob běhu programu také není zrovna nejrychlejší.




Jaké může mít tedy tento postup **výhody**? Je jich hned několik:

- **Přenositelnost**: Program je plně přenositelný, pokud existuje interpret pro danou platformu, půjde tam zdrojový kód programu spustit (a vývoj interpretu je snažší, než vývoj kompilátoru).
- **Jednodušší vývoj** - Ve vyšších jazycích jsme odstíněni od správy paměti, kterou za nás dělá tzv. garbage collector (řekneme si o něm v seriálu více). Často také nemusíme ani zadávat datové typy a máme k dispozici vysoce komfortní kolekce a další struktury.
- **Stabilita** - Díky tomu, že interpret kódu rozumí, předejde chybám, které by zkompilovaný program jinak klidně vykonal. Běh interpretovaných programů je tedy určitě bezpečnější, dále umožňuje zajímavou vlastnost, tzv. reflexi, kdy program za běhu zkoumá sám sebe, ale o tom později.
- **Jednoduchá editace** - Program můžeme vyvíjet po částech a nahrávat na cílové umístění, díky tomu, že se nemusí kompilovat, ho je možné jednoduše editovat "za běhu".

Interpret má tři zásadní **nevýhody**:

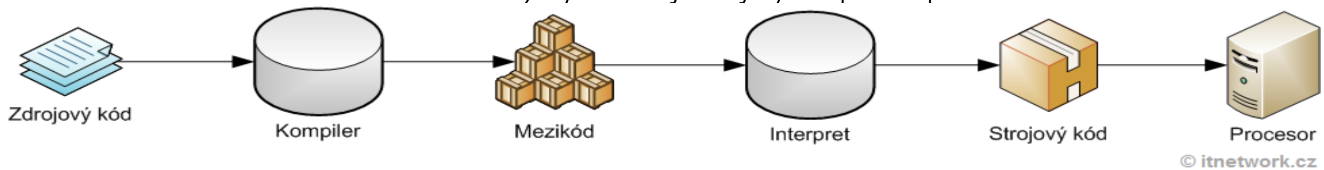
- **Rychlost** - Interpretace může být mnohdy velmi pomalá a program tak plně nevyužívá výkon počítače.
- **Často obtížné hledání chyb** - Díky kompilaci za běhu se chyby v kódu objeví až v tu chvíli, kdy je kód spuštěn. To může být někdy velmi nepříjemné.
- **Zranitelnost** - Protože se program šíří v podobě zdrojového kódu, každý do něj může zasahovat nebo krást jeho části.

Příkladem interpretovaného jazyka je např. PHP. Na většině webů ten poměrně pohodlný jazyk výkonově stačí, ale například Facebook používá speciální kompilovanou verzi PHP, zájemci ať se podívají na projekt HipHop for PHP.

Jazyky s virtuálním strojem

Napadlo vás, co by se stalo, kdyby se oba dva výše zmíněné způsoby spojily? Pokud ano, gratuluji, vynalezli jste virtuální stroj. Jedná se o nejmodernější podobu jazyka, která je v současné době také nejrozšířenější a nejlepší volbou pro vývoj většiny aplikací. Nebudu tajit, že do této kategorie spadá samotná Java nebo C#.

Zdrojový kód je nejprve přeložen do tzv. mezikódu, kterému se u Javy říká bytecode. Jedná se v podstatě o strojový (binární) kód, který má ale o poznání jednodušší instrukční sadu a přímo podporuje objektové programování. Tento mezikód je potom díky jednoduchosti relativně rychle interpretovatelný tzv. virtuálním strojem (tedy interpretem, v případě Javy je to tzv. JVM - Java Virtual Machine). Výsledkem je strojový kód pro náš procesor.




Určitě jste trochu vyděšeni, ale věřte, že jsme v podstatě odstranili nevýhody interpreta i kompileru a můžeme využívat mnohé z jejich **výhod**:

- **Odhalení chyb ve zdrojovém kódu** - Díky kompilaci do bytekódu jednoduše odhalíme chyby ve zdrojovém kódu.
- **Stabilita** - Díky tomu, že interpret kódu rozumí, zastaví nás před vykonáním nebezpečné operace a na chybu upozorní. Můžeme také provádět reflexi (i když pro bytekód, ale od toho jsme většinou odstíněni).
- **Jednoduchý vývoj** - Máme k dispozici hitech datové struktury a knihovny, správu paměti za nás provádí garbage collector.
- **Slušná rychlost** - Rychlost se u virtuálního stroje pohybuje mezi interpretem a kompilerem. Virtuální stroj již výsledky své práce po použití nezahazuje, ale dokáže je cachovat, sám se tedy optimalizuje při četnějších výpočtech a může dosahovat až rychlosti kompileru. Start programu bývá pomalejší, protože stroj překládá společně využívané knihovny.
- **Málo zranitelný kód** - Aplikace se šíří jako zdrojový kód v bytekódu, není tedy úplně jednoduše lidsky čitelná.
- **Přenositelnost** - Asi je jasné, že hotový program poběží na každém železe, na kterém se nachází virtuální stroj.

Java a JDK

Java je distribuována ve třech edicích:

- Java SE - Standardní Edici budeme používat pro začátek
- Java EE - Enterprise Edice není ve skutečnosti jiná Java, ale sada knihoven do JSE, která umožňuje vytvářet velké webové aplikace. Je poměrně komplikovaná, ale ve firmách extrémně žádaná. Pokud se ji naučíte, budete velmi žádaní programátoři.
- Java ME - Mikro Edice běží v SIM kartách, pračkách a dalších elektronických zařízeních (Oracle tvrdí, že Java pohání 9 miliard zařízení).

**Pro spuštění** našich aplikací budeme potřebovat **JRE** (Java Runtime Environment), což je běhové prostředí obsahující virtuální stroj. **Pro vývoj** budeme potřebovat **JDK** (Java Development Kit), které obsahuje knihovny a nástroje pro vývojáře.

Další výhodou Javy je, že je zcela zdarma a tedy dostupná všem vývojářům. Aplikace v Javě lze také spouštět přímo ve webovém prohlížeči pomocí Java Web Start. Ten se také automaticky stará o aktualizaci vaší aplikace.
Jazyky s virtuálním strojem ctí objektově orientované programování a jedná se o současný vrchol vývoje v této oblasti. Existují i jazyky 4. a 5. generace, ale ty mají specifické použití a nebudeme se s nimi zde zatím zabývat.
Nyní víme, s čím to vlastně budeme pracovat. Příště si ukážeme práci s IDE (programátorským prostředím) NetBeans a vytvoříme si svůj první program.

## 2. díl - NetBeans IDE a první konzolová aplikace

V minulém tutoriálu jsme si řekli něco o jazyce jako takovém a také jsme pochopili, jak Java funguje. Dnes se zaměříme hlavně na IDE NetBeans, ukážeme si, jak se instaluje, používá a naprogramujeme si jednoduchou konzolovou aplikaci.

Instalace

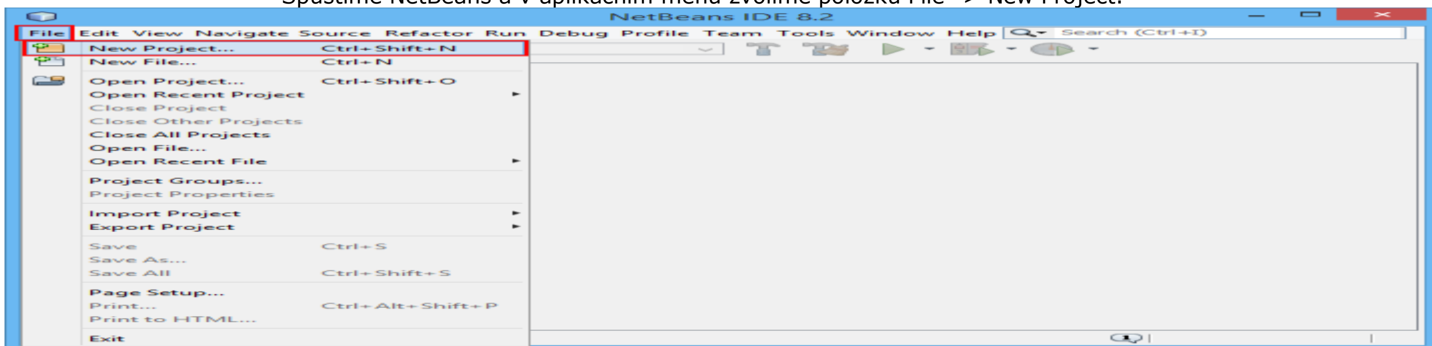
Nejprve si musíme stáhnout tzv. JDK (Java Development Kit), je to soubor základních nástrojů, které potřebujeme pro vývoj v Javě. Ten nalezneme na http://www.oracle.com/…s/index.html . Na stránce zaškrtneme, že souhlasíme s licencí, a vybereme JDK pro váš operační systém (nejčastěji Windows 32bit (označena jako x86) nebo 64bit, podle vaší verze operačního systému). Instalaci odklikáme (vynextíme), není třeba nic nastavovat. Je možné, že se s instalací spustí instalace dalších komponent, ty také odklikáme.
Nyní stáhneme samotné IDE. IDE je zkratka Integrated Development Environment (integrované vývojové prostředí) a jednoduše řečeno se jedná o aplikaci, ve které píšeme zdrojový kód a pomocí které potom naši aplikaci testujeme a ladíme. My budeme používat NetBeans, jelikož je asi nejrozšířenější a hlavně ho vyvíjí ta samá firma, která má na svědomí vývoj Javy samotné. Alternativou k NetBeans je ještě Eclipse.
Na adrese http://netbeans.org/…s/index.html stáhneme verzi pro vývoj v Java SE, která značí Java Standard Edition. Nainstalujeme (instalaci opět odklikáme) a spustíme.

Zálohování a verzování

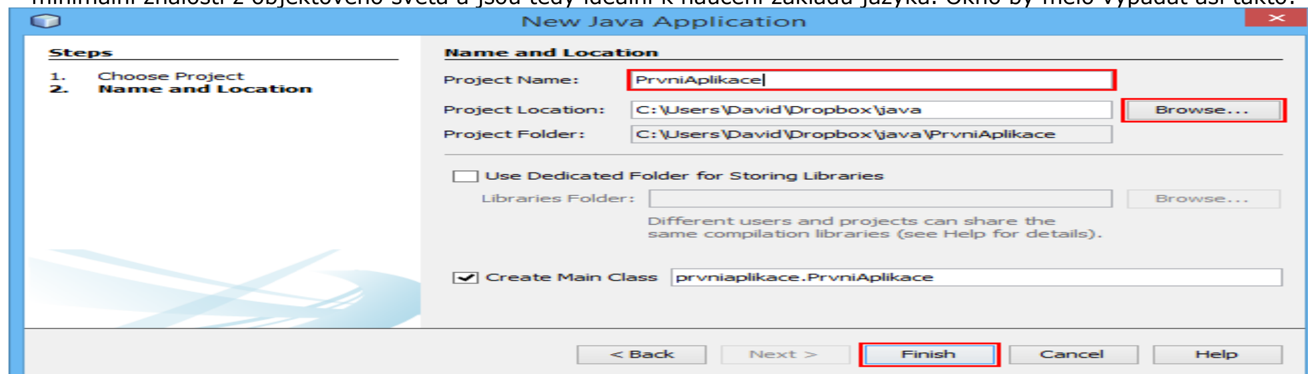
Kromě IDE programátor potřebuje nějaký nástroj, který bude zálohovat a verzovat jeho práci. Nemůžeme se spolehnout na to, že program prostě budeme ukládat, protože jsme lidé a ne stroje. Lidé dělají chyby a když přijdete o několikadenní nebo dokonce několikatýdenní práci, může to zabolet. Je dobré naučit se na toto myslet hned od začátku. Velmi doporučuji program DropBox, který je extrémně jednoduchý a sám vaše soubory **verzuje** (tedy zachovává změny v čase a je možné se vrátit ke starším verzím projektu) a zároveň **synchronizuje** s webovým úložištěm, i kdyby jste si projekt omylem smazali, přepsali, ukradli vám notebook nebo vám zkolaboval pevný disk, vaše data zůstanou v bezpečí. DropBox také umožňuje sdílet jeden projekt mezi více vývojáři. Více o DropBoxu viz tento článek, který obsahuje zároveň pozvánku do DropBoxu s 0,5 GB prostoru navíc.
Jako další verzovací nástroj se hojně používá GIT, jeho nastavení by ale vydalo na samostatný článek a DropBox pro naše účely bohatě postačuje.

Vytvoření projektu

Spustíme NetBeans a v aplikačním menu zvolíme položku File -> New Project.



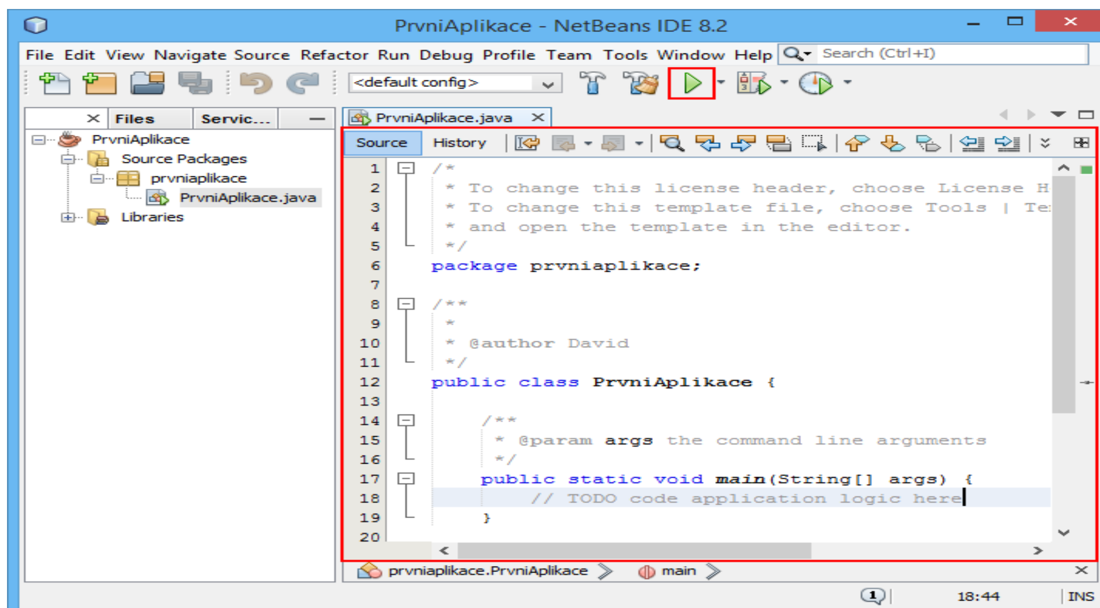

V okně New project vybereme z Java - Java Application. Jako jméno aplikace zvolíme PrvniAplikace. V Dropboxu si vytvořte nějakou složku na vaše projekty, např. Java. U lokace pomocí tlačítka Browse vybereme složku C:\Users\vase_jme-no\Dropbox\Java. Nějakou dobu zůstaneme u konzolových aplikací (příkazová řádka), protože k jejich obsluze potřebujeme minimální znalosti z objektového světa a jsou tedy ideální k naučení základů jazyka. Okno by mělo vypadat asi takto:




Formulář potvrdíme.

Ovládání NetBeans

V NetBeans se nám založil nový projekt. Já jsem okno hodně zmenšil, aby se mi sem vešlo 🙂




Zajímat nás bude zejména prostřední okno, do kterého nám NetBeans vygeneroval kostru zdrojového kódu. Možná může být překvapením, že nezačínáme s prázdným oknem, ale rovnou s kusem kódu. Proč tomu tak je pochopíte, až si kód alespoň intuitivně vysvětlíme, vše bude vysvětleno během seriálu a některé části jsou na pochopení poměrně složité, proto nám zatím bude stačit vědět, že tam prostě jsou.

Package a class zatím nebudeme řešit, spokojíme se s tím, že je to určitý způsob, jak se aplikace v Javě strukturují. Klíčová pro nás bude metoda **main**, mezi ty složené závorky pod ní (tedy do jejího těla) budeme psát náš kód. Main je vyhrazené slovo a Java ví, že má po spuštění aplikace vykonat právě tuto metodu (může jich tam být totiž více, ale o tom opět později). Vlastně můžeme zatím ignorovat úplně všechno až na tělo metody main.

Druhým důležitým prvkem v okně pro nás bude zelené tlačítko Play v horní liště, které program zkompiluje a spustí. Můžete si to zkusit, protože náš program zatím nic nedělá, hned se zase vypne. Spuštění můžeme provést též klávesovou zkratkou F6.

Hello world

Je zarytým zvykem, že prvním programem v nějakém novém jazyce bývá tzv. Hello world. Jedná se o program, který jakýmkoli způsobem uživateli zobrazí hlášku "Hello world", případně nějaký podobný text. Opět zopakuji, že příkazy budeme psát do těla metody main.

K výpisu textu slouží:

```
System.out.println("Text");
```

System je tzv. **třída**. Pojmem třída budeme zatím chápat soubor nějakých příkazů, příkazům se v Javě říká metody. System tedy obsahuje metody k obsluze vstupů a výstupů. Na výstupu (out) voláme metodu println, která vypíše text. Vidíme, že metodu na třídě voláme pomocí operátoru tečka. Každá metoda může obsahovat nějaké vstupní parametry, které se zadávají do závorky a jsou oddělené čárkou. V případě metody println je parametrem text k vypsání. Textu budeme říkat textový řetězec nebo jen řetězec (anglicky string) a budeme ho psát do uvozovek, aby tomu Java rozuměla a nezaměňovala ho s jinými příkazy. I kdyby metoda neměla žádné parametry, je závorka za ní povinná a byla by prázdná. Příkazy píšeme na samostatné řádky a za každý píšeme středník. Naše metoda main tedy bude nyní vypadat nějak takto:

Spustit kód

Klikni pro editaci

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

Program spustíme pomocí klávesy F6.

PrvniAplikace

Hello ITnetwork!

Gratuluji, právě jste se stali programátorem 🙂 To bude pro dnešek vše, příště se podíváme na [základní datové typy](#) a vytvoříme si jednoduchou kalkulačku.

Dnešní projekt je přiložen jako soubor na konci článku, i u dalších tutoriálů budu vždy výsledek přikládat ke stažení. Doporučuji si ale nejprve projekt vytvořit pomocí tutoriálu a ke stažení se uchýlit jen v případě, když vám něco nepůjde. Pokud program hned jen stáhnete, nic se nenaučíte 🙂

## 3. díl - Proměnné, typový systém a parsování

[Z minulého tutoriálu o Javě](#) již umíme pracovat s NetBeans a vytvořit konzolovou aplikaci. Dnes se podíváme na tzv. typový systém, ukážeme si základní datové typy a práci s proměnnými. Výsledkem budou 4 jednoduché programy včetně kalkulačky.

Proměnné

Než začneme řešit datové typy, pojďme se shodnout na tom, co je to proměnná (programátoři mi teď jistě odpustí zbytečné vysvětlování). Určitě znáte z matematiky proměnnou (např. x), do které jsme si mohli uložit nějakou hodnotu, nejčastěji číslo. Proměnná je v informatice naprosto to samé, je to místo v paměti počítače, kam si můžeme uložit nějaká data (jméno uživatele, aktuální čas nebo databázi článků). Toto místo má podle typu proměnné také vyhrazenou určitou velikost, kterou proměnná nesmí přesahnout (např. číslo nesmí být větší než 2 147 483 647).

Proměnná má vždy nějaký **datový typ**, může to být číslo, znak, text a podobně, záleží na tom, k čemu ji chceme používat. Většinou musíme před prací s proměnnou tuto proměnnou nejdříve tzv. deklarovat, čili říci jazyku jak se bude jmenovat a jakého datového typu bude (jaký v ní bude obsah). Jazyk ji v paměti založí a teprve potom s ní můžeme pracovat. Podle datového typu proměnné si ji jazyk dokáže z paměti načíst, modifikovat, případně ji v paměti založit. O každém datovém typu jazyk ví, kolik v paměti zabírá místa a jak s tímto kusem paměti pracovat.

Typový systém

Existují dva základní **typové systémy**: **statický** a **dynamický**.
**Dynamický typový systém** nás plně odstiňuje od toho, že proměnná má vůbec nějaký datový typ. Ona ho samozřejmě vnitřně má, ale jazyk to nedává nejevo. Dynamické typování jde mnohdy tak daleko, že proměnné nemusíme ani deklarovat, jakmile do nějaké proměnné něco uložíme a jazyk zjistí, že nebyla nikdy deklarována, sám ji založí. Do té samé proměnné můžeme ukládat text, potom objekt uživatele a potom desetinné číslo. Jazyk se s tím sám popere a vnitřně automaticky mění datový typ. V těchto jazycích jde vývoj rychleji díky menšímu množství kódu, zástupci jsou např. PHP nebo Ruby.
**Statický typový systém** naopak striktně vyžaduje definovat typ proměnné a tento typ je dále neměnný. Jakmile proměnnou jednou deklarujeme, není možné její datový typ změnit. Jakmile se do textového řetězce pokusíme uložit objekt uživatel, dostaneme vynadáno.
**Java je staticky typovaný jazyk**, všechny proměnné musíme nejprve deklarovat s jejich datovým typem. Nevýhodou je, že díky deklaracím je zdrojový kód poněkud objemnější a vývoj pomalejší. Obrovskou výhodou však je, že nám kompiler před spuštěním zkontroluje, zda všechny datové typy sedí. Dynamické typování sice vypadá jako výhodné, ale zdrojový kód není možné automaticky kontrolovat a když někde očekáváme objekt uživatel a přijde nám tam místo toho desetinné číslo, odhalí se chyba až za běhu a interpret program shodí. Naopak Java nám nedovolí program ani zkompilovat a na chybu nás upozorní (to je další výhoda kompilace).
Pojďme si nyní něco naprogramovat, ať si nabyté znalosti trochu osvojíme, s teorií budeme pokračovat až příště. Řekněme si nyní tři základní datové typy:

- Celá čísla: **int**
- Desetinná čísla: **float**
- Textový řetězec: **String**

String píšeme s velkým písmenem na začátku, časem se dozvíme proč.
Program vypisující proměnnou
Zkusíme si nadeklarovat celočíselnou proměnnou *a*, dosadit do ní číslo 56 a její obsah vypsat do konzole. Založte si nový projekt, pojmenujte ho Vypis (i ke všem dalším příkladům si vždy založte nový projekt). Kód samozřejmě píšeme do těla metody main (jako minule), čili ji zde již nebudu opisovat.
Spustit kód
Klikni pro editaci

```
int a;
a = 56;
System.out.println(a);
```

První příkaz nám nadeklaruje novou proměnnou *a* datového typu int, proměnná tedy bude sloužit pro ukládání celých čísel. Druhý příkaz provádí přiřazení do proměnné, slouží k tomu operátor "rovná se". Poslední příkaz je nám známý, vypíše do konzole obsah proměnné *a*. Konzole je chytrá a umí vypsat i číselné proměnné.
Konzolová aplikace
56
Pro desetinnou proměnnou by kód vypadal takto:
Spustit kód
Klikni pro editaci

```
float a;
a = 56.6F;
System.out.println(a);
```

Je to téměř stejné jako s celočíselným. Jako desetinný oddělovač používáme tečku a na konci desetinného čísla je nutné zadat tzv. suffix F, tím říkáme, že se jedná o float.
Program Papoušek
Minulý program byl poněkud nudný, zkusme nějak reagovat na vstup od uživatele. Napíšeme program papoušek, který bude dvakrát opakovat to, co uživatel napsal. Ještě jsme nezkoušeli z konzole nic načítat. Slouží k tomu metoda nextLine(), která uživateli umožní zadat do konzole řádku textu a nám vrátí zadaný textový řetězec. Abychom mohli z konzole načítat, založíme si nový projekt s názvem Papousek a jeho kód upravíme tak, aby vypadal takto:

```
package papousek;

import java.util.Scanner;

public class Papousek {

public static void main(String[] args)
{
Scanner sc = new Scanner(System.in, "Windows-1250");

}
}
```

Pro názornost jsem vymazal šedou dokumentaci, ale klidně si ji tam nechte. Změna spočívá v importování java.util.Scanner, což nám umožňuje přístup k metodám pro vstup z konzole. Konečně ten dlouhý řádek na začátku metody nedělá nic jiného, než že nám vytvoří proměnnou sc, na které můžeme volat onu metodu nextLine(), která načte z konzole další řetězec. Vysvětlit si ho by bylo nyní příliš komplikované, berte ho jako že tam je, časem ho pochopíme.
**Pokud budete potřebovat v kterémkoli ze svých programů načíst text z konzole, je nutné program takto upravit a přidat proměnnou sc!**
Nyní se přesuňme k samotnému kódu programu a upravme obsah metody main() do následující podoby:
Spustit kód
Klikni pro editaci

```
Scanner sc = new Scanner(System.in, "Windows-1250");
System.out.println("Ahoj, jsem virtuální papoušek Lóra, rád opakuji!");
System.out.println("Napiš něco: ");
String vstup;
vstup = sc.nextLine();
String vystup;
```

```
vystup = vstup + ", " + vstup + "!";
System.out.println(vystup);
```

To už je trochu zábavnější 🙂 První řádek jsme ji již vysvětlili výše, další dva řádky jsou jasné (vypisují text). Dále deklarujeme textový řetězec *vstup*. Do *vstup* se přiřadí hodnota z metody nextLine() na konzoli, tedy to, co uživatel zadal. Pro výstup si pro názornost zakládáme další proměnnou typu textový řetězec. Zajímavé je, jak do *vystup*přiřadíme, tam využíváme tzv. konkatenace (spojování) řetězců. Pomocí operátoru "+" totiž můžeme spojit několik textových řetězců do jednoho a je jedno, jestli je řetězec v proměnné nebo je explicitně zadán v uvozovkách ve zdrojovém kódu. Do proměnné tedy přiřadíme vstup, dále čárku, znovu vstup a poté vykřičník. Proměnnou vypíšeme a skončíme.

Konzolová aplikace

```
Ahoj, jsem virtuální papoušek Lóra, rád opakuji!
Napiš něco:
Nazdar ptáku
Nazdar ptáku, Nazdar ptáku!
```

Do proměnné můžeme přiřazovat již v její deklaraci, můžeme tedy nahradit:

```
String vstup;
vstup = sc.nextLine();
```

za

```
String vstup = sc.nextLine();
```

Program by šel zkrátit ještě více v mnoha ohledech, ale obecně je lepší používat více proměnných a dodržovat přehlednost, než psát co nejkratší kód a po měsíci zapomenout, jak vůbec funguje.

Program zdvojnásobovač

Zdvojnásobovač si vyžádá na vstupu číslo a to poté zdvojnásobí a vypíše. Asi bychom s dosavadními znalostmi napsali něco takového:

```
Scanner sc = new Scanner(System.in, "Windows-1250");
System.out.println("Zadejte číslo k zdvojnásobení:");
int a = sc.nextLine();
a = a * 2;
System.out.println(a);
```

Všimněte si zdvojnásobení čísla *a*, které jsme provedli pomocí přiřazení. Java nám nyní vyhubuje a podtrhne řádek, ve kterém se snažíme hodnotu z konzole dostat do proměnné typu int. Narážíme na typovou kontrolu, konkrétně nám nextLine() vrací String a my se ho snažíme uložit do intu. Budeme ho potřebovat tzv. **naparsovat**.

Parsování

Parsováním se myslí převod z textové podoby na nějaký specifický typ, např. číslo. Mnoho datových typů má v Javě již připraveny metody k parsování, pokud budeme chtít naparsovat např. int ze Stringu, budeme postupovat takto:

```
String s = "56";
int a = Integer.parseInt(s);
```

Metoda parseInt() se volá na třídě Integer, bere jako parametr textový řetězec a vrátí číslo. Využijeme této znalosti v našem programu:

Spustit kód

Klikni pro editaci

```
Scanner sc = new Scanner(System.in, "Windows-1250");
System.out.println("Zadejte číslo k zdvojnásobení:");
String s = sc.nextLine();
int a = Integer.parseInt(s);
a = a * 2;
System.out.println(a);
```

Nejprve si text z konzole uložíme do textového řetězce *s*. Do celočíselné proměnné *a* poté uložíme číselnou hodnotu řetězce *s*. Dále hodnotu v *a* zdvojnásobíme a vypíšeme do konzole.

Konzolová aplikace

```
Zadejte číslo k zdvojnásobení:
1024
2048
```

Parsování se samozřejmě nemusí povést, když bude v textu místo čísla např. slovo, ale tento případ zatím nebudeme ošetřovat.

Jednoduchá kalkulačka

Ještě jsme nepracovali s desetinnými čísly, zkusme si napsat slibovanou kalkulačku. Bude velmi jednoduchá, na vstup přijdou dvě čísla, program poté vypíše výsledky pro sčítání, odčítání, násobení a dělení.

Spustit kód

Klikni pro editaci

```
Scanner sc = new Scanner(System.in, "Windows-1250");
System.out.println("Vítejte v kalkulačce");
System.out.println("Zadejte první číslo:");
float a = Float.parseFloat(sc.nextLine());
System.out.println("Zadejte druhé číslo:");
float b = Float.parseFloat(sc.nextLine());
float soucet = a + b;
float rozdil = a - b;
float soucin = a * b;
float podil = a / b;
System.out.println("Součet: " + soucet);
System.out.println("Rozdíl: " + rozdil);
System.out.println("Součin: " + soucin);
System.out.println("Podíl: " + podil);
System.out.println("Děkuji za použití kalkulačky, aplikaci ukončíte libovolnou klávesou.");
```

Konzolová aplikace

```
Vítejte v kalkulačce
```

Zadejte první číslo:
3.14
Zadejte druhé číslo:
2.72
Součet: 5.86
Rozdíl: 0.42
Součin: 8.5408
Podíl: 1.15441176470588
Děkuji za použití kalkulačky, aplikaci ukončíte libovolnou klávesou.

Všimněte si dvou věcí. Zaprvé jsme zjednodušili parsování z konzole tak, abychom nepotřebovali Stringovou proměnnou, protože bychom ji stejně již poté nepoužili. Zadruhé na konci programu spojujeme řetězec s číslem pomocí znaménka plus. Java překvapivě nezahlásí chybu, ale provede tzv. implicitní konverzi a zavolá na čísle metodu Integer.toString() nebo přímo na Stringu String.valueOf(). Kdyby tomu tak nebylo nebo jsme se dostali do situace, kdy potřebujeme převést cokoli na String, zavoláme metodu String.valueOf() a jako parametr ji dáme naši proměnnou. Java to v tomto případě udělala za nás, v podstatě vykoná:

```
System.out.println("Součet: " + String.valueOf(soucet));
```

Právě jsme se tedy naučili opak k parsování - převést cokoli do textové podoby. Příště si řekneme více o typovém systému v Javě a představíme si další datové typy.

Všechny programy máte samozřejmě opět v příloze, zkoušejte si vytvářet nějaké podobné, znalosti již k tomu máte

## 4. díl - Typový systém podruhé: Datové typy

V minulém dílu seriálu jsme si ukázali základní datové typy, byly to int, String a float. Nyní se na datové typy podíváme více zblízka a vysvětlíme si, kdy jaký použít. Dnešní lekce bude hodně teoretická, ale o to více bude praktická ta příští. Na konci si vytvoříme pár jednoduchých ukázek.

Java rozeznává dva druhy datových typů, **primitivní** a **referenční**.

Primitivní datové typy

Proměnné primitivního datového typu si dokážeme jednoduše představit. Může se jednat např. o číslo nebo znak. V paměti je jednoduše uložena přímo hodnota a my k této hodnotě můžeme z programu přímo přistupovat. Slovo přímo jsem tolikrát nepoužil jen náhodou. V této sekci tutoriálů se budeme věnovat výhradně těmto proměnným.

Celočíselné datové typy

Podívejme se nyní na tabulku všech vestavěných celočíselných datových typů v Javě, všimněte si typu int, který již známe zminule.

| Datový typ | Rozsah | V |
|---|---|---|
| byte | -128 až 127 | |
| short | -32 768 až 32 767 | 1 |
| **int** | -2 147 483 648 až 2 147 483 647 | 3 |
| long | -9 223 372 036 854 775 808 až 9 223 372 036 854 775 807 | 6 |

Asi vás napadá otázka, proč máme tolik možných typů pro uložení čísla. Odpověď je prostá, záleží na jeho velikosti. Čím větší číslo, tím více spotřebuje paměti. Pro věk uživatele tedy zvolíme byte, protože se jistě nedožije více, než 127 let. Představte si databázi milionu uživatelů nějakého systému, když zvolíme místo byte int, bude zabírat 4x více místa. Naopak když budeme mít funkci k výpočtu faktoriálu, stěží nám bude stačit rozsah intu a použijeme long.

Nad výběrem datového typu nemusíte moc přemýšlet a většinou se používá jednoduše int. Typ řešte pouze v případě, když jsou proměnné v nějakém poli (obecně kolekci) a je jich tedy více, potom se vyplatí zabývat se paměťovými nároky. Tabulky sem dávám spíše pro úplnost. Mezi typy samozřejmě funguje již zmíněná implicitní konverze, tedy můžeme přímo přiřadit int do proměnné typu long a podobně, aniž bychom něco konvertovali.

Desetinná čísla

U desetinných čísel je situace poněkud jednodušší, máme na výběr pouze dva datové typy. Samozřejmě se liší opět v rozsahu hodnoty, dále však ještě v přesnosti (vlastně počtu des. míst). Double má již dle názvu dvojnásobnou přesnost oproti float.

| Datový typ | Rozsah | Přesnos |
|---|---|---|
| float | +-1.5 * $10^{-45}$ až +-3.4 * $10^{38}$ | 7 čísel |
| **double** | +-5.0 * $10^{-324}$ až +-1.7 * $10^{308}$ | 15-16 čís |

*Pozor, vzhledem k tomu, že desetinná čísla jsou v počítači uložena ve dvojkové soustavě, dochází k určité ztrátě přesnosti. Odchylka je sice téměř zanedbatelná, nicméně když budete programovat např. finanční systém, nepoužívejte tyto dat. typy pro uchování peněz, mohlo by dojít k malým odchylkám.*

Když do floatu chceme dosadit přímo ve zdrojovém kódu, musíme použít sufix F, u double sufix D (u double ho můžeme vypustit, jelikož je výchozí desetinný typ):

```
float f = 3.14F;
double d = 2.72;
```

Jako desetinný separátor používáme ve zdrojovém kódu vždy tečku, nehledě na to, jaké máme v operačním systému regionální nastavení.

Další vestavěné datové typy

Podívejme se na další datové typy, které nám Java nabízí:

| Datový typ | Rozsah | Velikost/Přesnost |
|---|---|---|
| char | U+0000 až U+ffff | 16 bitů |
| boolean | true nebo false | 8 bitů |

**Char**

Char nám reprezentuje jeden znak, narozdíl od Stringu, který reprezentoval celý řetězec charů. Znaky v Javě píšeme do apostrofů:

```
char c = 'A';
```

Char patří v podstatě do celočíselných proměnných (obsahuje číselný kód znaku), ale přišlo mi logičtější uvést ho zde.

**BigDecimal**

Typ BigDecimal řeší problém ukládání desetinných čísel v binární podobě, ukládá totiž číslo vnitřně jako pole. Používá se tedy pro uchování peněžních hodnot. Nebudeme si zde ukazovat použití, protože se používá jako objekt. Pokud budete dělat někdy finanční výpočty, tak si na něj vzpomeňte.

*Pozn.: V Javě jsou čísla tzv. odděděna od třídy Number. To je spíše informace do budoucna. Jelikož nyní nevíme, co dědičnost znamená, důležitá pro nás není. Number obsahuje ještě čtyři podtřídy, kterými se nebudeme podrobněji zabývat. BigDecimal a BigInteger slouží k výpočtům s vysokou přesností. Třídy AtomicInteger a AtomicLong se používají v aplikacích s více podprocesy. Opět je důležité, abyste věděli, že něco takového existuje a případně se sem později vrátili.*

**Boolean**

Boolean nabývá dvou hodnot: true (pravda) a false (nepravda). Budeme ho používat zejména tehdy, až se dostaneme k podmínkám. Do proměnné typu boolean lze uložit jak přímo hodnotu true/false, tak i logický výraz. Zkusme si jednoduchý příklad:

Spustit kód

Klikni pro editaci

```
boolean b = false;
boolean vyraz = (15 > 5);
System.out.println(b);
System.out.println(vyraz);
```

Výstup programu:

Konzolová aplikace

```
false
true
```

Výrazy píšeme do závorek. Vidíme, že výraz nabývá hodnoty true (pravda), protože 15 je opravdu větší než 5. Od výrazů je to jen krok k podmínkám, na ně se podíváme příště.

Referenční datové typy

K referenčním typům se dostaneme až u objektově orientovaného programování, kde si také vysvětlíme zásadní rozdíly. Zatím budeme pracovat jen s tak jednoduchými typy, že rozdíl nepoznáme. Spokojíme se s tím, že referenční typy jsou složitější, než ty primitivní. Jeden takový typ již známe, je jím String. Možná vás napadá, že String nemá nijak omezenou délku, je to tím, že s referenčními typy se v paměti pracuje jinak. Hodnotové typy začínají narozdíl od typů primitivních velkým písmenem.

String má na sobě řadu opravdu užitečných metod. Některé si teď probereme a vyzkoušíme:

String

**StartsWith(), endsWith() a contains()**

Můžeme se jednoduše zeptat, zda řetězec začíná, končí nebo zda obsahuje určitý podřetězec (substring). Podřetězcem myslíme část původního řetězce. Všechny tyto metody budou jako parametr brát samozřejmě podřetězec a vracet hodnoty typu boolean (true/false). Zatím na výstup neumíme reagovat, ale pojďme si ho alespoň vypsat:

Spustit kód

Klikni pro editaci

```
String s = "Krokonosohroch";
System.out.println(s.startsWith("krok"));
System.out.println(s.endsWith("hroch"));
System.out.println(s.contains("nos"));
System.out.println(s.contains("roh"));
```

Výstup programu:

Konzolová aplikace

```
false
true
true
false
```

Vidíme, že vše funguje podle očekávání. První výraz samozřejmě neprošel díky tomu, že řetězec ve skutečnosti začíná velkým písmenem.

**toUpperCase() a toLowerCase()**

Rozlišování velkých a malých písmen může být někdy na obtíž. Mnohdy se budeme potřebovat zeptat na přítomnost podřetězce tak, aby nezáleželo na velikosti písmen. Situaci můžeme vyřešit pomocí metod toUpperCase() a toLowerCase(), které vrací řetězec ve velkých a v malých písmenech. Uveďme si reálnější příklad než je Krokonosohroch. Budeme mít v proměnné řádek konfiguračního souboru, kterou psal uživatel. Jelikož se na vstupy od uživatelů nelze spolehnout, musíme se snažit eliminovat možné chyby, zde např. s velkými písmeny.

Spustit kód

Klikni pro editaci

```
String konfig = "Fullscreen shaDows autosave";
konfig = konfig.toLowerCase();
System.out.println("Poběží hra ve fullscreenu?");
System.out.println(konfig.contains("fullscreen"));
System.out.println("Budou zapnuté stíny?");
System.out.println(konfig.contains("shadows"));
System.out.println("Přeje si hráč vypnout zvuk?");
System.out.println(konfig.contains("nosound"));
System.out.println("Přeje si hráč hru automaticky ukládat?");
System.out.println(konfig.contains("autosave"));
```

Výstup programu:

Konzolová aplikace

```
Poběží hra ve fullscreenu?
```

true
Budou zapnuté stíny?
true
Přeje si hráč vypnout zvuk?
false
Přeje si hráč hru automaticky ukládat?
true

Vidíme, že jsme schopni zjistit přítomnost jednotlivých slov v řetězci tak, že si nejprve řetězec převedeme celý na malá písmena (nebo na velká) a potom kontrolujeme přítomnost slova jen malými (nebo velkými) písmeny. Takhle by mimochodem mohlo opravdu vypadat jednoduché zpracování nějakého konfiguračního skriptu.

**Trim()**

Další nástrahou mohou být mezery a obecně všechny tzv. bílé znaky, které nejsou vidět, ale mohou nám uškodit. Obecně může být dobré trimovat všechny vstupy od uživatele. Zkuste si v následující aplikaci před číslo a za číslo zadat několik mezer, trim() je odstraní. Odstraňují se vždy bílé znaky kolem řetězce, nikoli uvnitř:

Spustit kód
Klikni pro editaci

```
System.out.println("Zadejte číslo:");
String s = sc.nextLine();
System.out.println("Zadal jste text: " + s);
System.out.println("Text po funkci trim: " + s.trim());
int a = Integer.parseInt(s.trim());
System.out.println("Převedl jsem zadaný text na číslo parsováním, zadal jste: " + a);
```

**Replace()**

Asi nejdůležitější metodou na Stringu je nahrazení určité jeho části jiným textem. Jako parametry zadáme dva podřetězce, jeden co chceme nahrazovat a druhý ten, kterým to chceme nahradit. Metoda vrátí nový String, ve kterém proběhlo nahrazení. Když daný podřetězec metoda nenajde, vrátí původní řetězec. Zkusme si to:

Spustit kód
Klikni pro editaci

```
String s = "C# je nejlepší!";
s = s.replace("C#", "Java");
System.out.println(s);
```

Výstup programu:
Konzolová aplikace
Java je nejlepší!

**Format()**

Format() je velmi užitečná metoda, která nám umožňuje vkládat do samotného textového řetězce zástupné značky. Ty jsou reprezentovány jako procento a zkratka datového typu. Metoda se volá na typu String, prvním parametrem je textový řetězec se značkami, další dále následují proměnné v tom pořadí, v kterém se mají do textu místo značek vložit. Všimněte si, že se metoda nevolá na konkrétní proměnné (přesněji instanci, viz další díly), ale přímo na typu String.

Spustit kód
Klikni pro editaci

```
int a = 10;
int b = 20;
int c = a + b;
String s = String.format("Když sečteme %d a %d, dostaneme %d", a, b, c);
System.out.println(s);
```

Výstup programu:
Konzolová aplikace
Když sečteme 10 a 20, dostaneme 30

Značky jsou:

- %d pro celá čísla
- %s pro Stringy
- %f pro float. U float můžeme definovat délku desetinné části, např: %.2f pro dvě desetinná místa.

Konzole sama umí přijímat text v takovémto formátu, jen musíme místo println() volat printf(). Můžeme tedy napsat:

Spustit kód
Klikni pro editaci

```
int a = 10;
int b = 20;
int c = a + b;
System.out.printf("Když sečteme %d a %d, dostaneme %d", a, b, c);
```

Toto je velmi užitečná a přehledná cesta, jak sestavovat řetězce, i přesto se však v Javě řetězce spojují spíše pomocí operátoru "+".

**Length()**

Poslední, ale nejdůležitější je length(), tedy délka. Vrací celé číslo, které představuje počet znaků v řetězci.

Spustit kód
Klikni pro editaci

```
System.out.println("Zadejte vaše jméno:");
String jmeno = sc.nextLine();
System.out.printf("Délka vašeho jména je: %d", jmeno.length());
```

Je toho ještě spoustu k vysvětlování a jsou další datové typy, které jsme neprobrali. Aby jsme však stále neprobírali jen teorii, ukážeme si již příště podmínky a cykly, potom bude naše programátorská výbava dostatečně velká k tomu, abychom tvořili zajímavé programy

**5. díl - Podmínky (větvení)**

V minulém dílu jsme si podrobně probrali datové typy. Abychom si něco naprogramovali, potřebujeme nějak reagovat na různé situace. Může to být například hodnota zadaná uživatelem, podle které budeme chtít měnit další běh programu. Říkáme, že se

program větví a k větvení používáme podmínky, těm se budeme věnovat celý dnešní díl. Vytvoříme program na výpočet odmocniny a vylepšíme naši kalkulačku.

Podmínky

V Javě se podmínky píší úplně stejně, jako ve všech CLike jazycích, pro začátečníky samozřejmě vysvětlím. Pokročilejší se asi budou chvilku nudit

Podmínky zapisujeme pomocí klíčového slova **if**, za kterým následuje logický výraz. Pokud je výraz pravdivý, provede se následující příkaz. Pokud ne, následující příkaz se přeskočí a pokračuje se až pod ním. Vyzkoušejme si to:

Spustit kód

Klikni pro editaci

```
if (15 > 5)
System.out.println("Pravda");
System.out.println("Program zde pokračuje dál");
```

Výstup programu:

Konzolová aplikace

Pravda

Program zde pokračuje dál

Pokud podmínka platí (což zde ano), provede se příkaz vypisující do konzole text pravda. V obou případech program pokračuje dál. Součástí výrazu samozřejmě může být i proměnná:

Spustit kód

Klikni pro editaci

```
System.out.println("Zadej nějaké číslo");
int a = Integer.parseInt(sc.nextLine());
if (a > 5)
System.out.println("Zadal jsi číslo větší než 5!");
System.out.println("Děkuji za zadání");
```

Ukažme si nyní relační operátory, které můžeme ve výrazech používat:

| **Operátor** | **C-like zápis** |
|---|---|
| Rovnost | == |
| Je ostře větší | > |
| Je ostře menší | < |
| Je větší nebo rovno | >= |
| Je menší nebo rovno | <= |
| Nerovnost | != |
| Obecná negace | ! |

Rovnost zapisujeme dvěma == proto, aby se to nepletlo s běžným přiřazením do proměnné, které se dělá jen jedním =. 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:

Spustit kód

Klikni pro editaci

```
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, to znamená, že ho mohu odmocnit!");
double o = Math.sqrt(a);
System.out.println("Odmocnina z čísla " + a + " je " + o);
}
System.out.println("Děkuji za zadání");
```

Konzolová aplikace

Zadej nějaké číslo, ze kterého spočítám odmocninu:

144

Zadal jsi číslo větší než 0, to znamená, že ho mohu odmocnit!

Odmocnina z čísla 144 je 12.0

Děkuji za zadání

Často můžete vidět použití bloku i v případě, že je pod podmínkou jen jeden příkaz, mnohdy je to totiž přehlednější.

Nezapomeňte si naimportovat java.util.Scanner, aby program znal třídu Scanner.

Program načte od uživatele číslo a pokud je větší než 0, vypočítá z něj druhou odmocninu. Mimo jiné jsme použili třídu Math, která na sobě obsahuje řadu užitečných matematických metod, někdy si ji blíže představíme. Sqrt() vrací hodnotu jako double.

Bylo by hezké, kdyby nám program vyhuboval v případě, že zadáme záporné číslo. S dosavadními znalostmi bychom napsali něco jako:

Spustit kód

Klikni pro editaci

```
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, to znamená, že ho mohu odmocnit!");
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");
```

Všimněte si, že musíme pokrýt i případ, kdy se a == 0, nejen když je menší. Kód však můžeme výrazně zjednodušit pomocí klíčového slova **else**, které vykoná následující příkaz nebo blok příkazů **v případě, že se podmínka neprovede**:

Spustit kód

Klikni pro editaci

```
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, to znamená, že ho mohu odmocnit!");
double o = Math.sqrt(a);
System.out.println("Odmocnina z čísla " + a + " je " + o);
}
else
{
System.out.println("Odmocnina ze záporného čísla neexistuje!");
}
System.out.println("Děkuji za zadání");
```

Kód je mnohem přehlednější a nemusíme vymýšlet opačnou podmínku, což by v případě složené podmínky mohlo být někdy i velmi obtížné. V případě více příkazů by byl za else opět blok { }.

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). Naivně bychom mohli kód napsat takto:

Spustit kód

Klikni pro editaci

```
int a = 0; // do a si přiřadíme na začátku 0

if (a == 0) // pokud je a 0, dáme do něj jedničku
{
a = 1;
}
if (a == 1) // pokud je a 1, dáme do něj nulu
{
a = 0;
}

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

Nefunguje to, že? Pojďme si projet, co bude program dělat. Na začátku máme v *a* nulu, první podmínka se jistě splní a dosadí do *a* jedničku. No ale rázem se splní i ta druhá. Co s tím? Když podmínky otočíme, budeme mít ten samý problém s jedničkou. Jak z toho ven? Ano, použijeme else.

Spustit kód

Klikni pro editaci

```
int a = 0; // do a si přiřadíme na začátku 0

if (a == 0) // pokud je a 0, dáme do něj jedničku
{
a = 1;
}
else // pokud je a 1, dáme do něj nulu
{
a = 0;
}

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

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

| Operátor | C-like Zápis |
| --- | --- |
| A zároveň | && |
| Nebo | \|\| |

Uveďme si příklad:

Spustit kód

Klikni pro editaci

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

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

S tím si zatím vystačíme, operátory se pomocí závorek samozřejmě dají kombinovat.

Spustit kód

Klikni pro editaci

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

Switch

Switch je konstrukce, převzatá z jazyka C (jako většina gramatiky Javy). Umožňuje nám zjednodušit (relativně) zápis více podmínek pod sebou. Vzpomeňme si na naši kalkulačku v prvních lekcích, která načetla 2 čísla a vypočítala všechny 4 operace. Nyní si ale budeme chtít zvolit, kterou operaci chceme. Bez switche bychom napsali kód podobný tomuto:

Spustit kód

Klikni pro editaci

```
Scanner sc = new Scanner(System.in, "Windows-1250");
System.out.println("Vítejte v kalkulačce");
System.out.println("Zadejte první číslo:");
float a = Float.parseFloat(sc.nextLine());
System.out.println("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;
if (volba == 1)
{
vysledek = a + b;
}
else if (volba == 2)
{
vysledek = a - b;
}
else if (volba == 3)
{
vysledek = a * b;
}
else if (volba == 4)
{
vysledek = a / b;
}
if ((volba > 0) && (volba < 5))
{
System.out.println("Výsledek: " + vysledek);
}
else
{
System.out.println("Neplatná volba");
}
System.out.println();
System.out.println("Děkuji za použití kalkulačky.");
```

```
Konzolová aplikace
Vítejte v kalkulačce
Zadejte první číslo:
3.14
Zadejte druhé číslo:
2.72
```

Zvolte si operaci:
1 - sčítání
2 - odčítání
3 - násobení
4 - dělení
2
Výsledek: 0.42
Děkuji za použití kalkulačky.

Všimněte si, že jsme proměnnou výsledek deklarovali na začátku, jen tak do ni 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 nám Java vyhubovala, že se snažíme vypsat proměnnou, která nebyla jednoznačně inicializována. Proměnná může být deklarována (založena v paměti) vždy jen jednou. Další vychytávka je kontrola správnosti volby. Program by v tomto případě fungoval stejně i bez těch else, ale nač se dále ptát, když již máme výsledek.

Nyní si zkusíme napsat ten samý kód pomocí switche:

Spustit kód

Klikni pro editaci

```
Scanner sc = new Scanner(System.in, "Windows-1250");
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.println("Výsledek: " + vysledek);
}
else
{
System.out.println("Neplatná volba");
}
System.out.println();
System.out.println("Děkuji za použití kalkulačky.");
```

Vidíme, že kód je trochu přehlednější. Pokud bychom potřebovali v nějaké větvi switche spustit více příkazů, překvapivě je nebudeme psát do bloku, ale rovnou pod sebe. Blok {} nám zde nahrazuje příkaz break, který způsobí vyskočení z celého switche. Switch může místo case x: obsahovat ještě možnost default:, která se vykoná v případě, že nebude platit žádný case. Je jen na vás, jestli budete switch používat, obecně se vyplatí jen při větším množství příkazů a vždy jde nahradit sekvencí if a else. Nezapomínejte na breaky. Switch je v Javě podporován i pro hodnoty stringové proměnné a to od Javy 7.

To bychom měli, příště nás čekají pole a cykly, tím dovršíme základní znalosti, máte se na co těšit

**6. díl - Cykly v Javě**

Minule jsme si v tutoriálu vysvětlili podmínky . Nyní přejdeme k cyklům, po dnešním tutoriálu již budeme mít téměř kompletní výbavu základních konstrukcí a budeme schopni tvořit rozumné aplikace.

Cykly

Jak již slovo cyklus napoví, něco se bude opakovat. Když chceme v programu něco udělat 100x, jistě nebudeme psát pod sebe 100x ten samý kód, ale vložíme ho do cyklu. Cyklů máme několik druhů, vysvětlíme si, kdy který použít. Samozřejmě si ukážeme praktické příklady.

FOR cyklus

Tento cyklus má stanovený **pevný počet opakování** a hlavně obsahuje tzv. řídící proměnnou (celočíselnou), ve které se postupně během běhu cyklu mění hodnoty. Syntaxe (zápis) cyklu for je následující:

```
for (promenna; podminka; prikaz)
```

- **promenna** je řídící proměnná cyklu, které nastavíme počáteční hodnotu (nejčastěji 0, protože v programování vše začíná od nuly, nikoli od jedničky). Např. tedy int i = 0. Samozřejmě si můžeme proměnnou i vytvořit někde nad tím a už nemusíme psát slovíčko int, bývá ale zvykem používat právě int i.

- **podminka** je podmínka vykonání dalšího kroku cyklu. Jakmile nebude platit, cyklus se ukončí. Podmínka může být např (i < 10).
- **prikaz** nám říká, co se má v každém kroku s řídící proměnnou stát. Tedy zda se má zvýšit nebo snížit. K tomu využijeme speciálních operátorů ++ a --, ty samozřejmě můžete používat i úplně běžně mimo cyklus, slouží ke zvýšení nebo snížení proměnné o 1.

Pojďme si udělat jednoduchý příklad, většina z nás jistě zná Sheldona z The Big Bang Theory. Pro ty co ne, budeme simulovat situaci, kdy klepe na dveře své sousedky. Vždy 3x zaklepe a poté zavolá: "Penny!". Náš kód by bez cyklů vypadal takto:

Spustit kód
Klikni pro editaci

```
System.out.println("Knock");
System.out.println("Knock");
System.out.println("Knock");
System.out.println("Penny!");
```

My ale už nic nemusíme otrocky opisovat:

Spustit kód
Klikni pro editaci

```
for (int i=0; i < 3; i++)
{
System.out.println("Knock");
}
System.out.println("Penny!");
```

Konzolová aplikace

```
Knock
Knock
Knock
Penny!
```

Cyklus proběhne 3x, zpočátku je v proměnné *i* nula, cyklus vypíše "Knock" a zvýší proměnnou *i* o jedna. Poté běží stejně s jedničkou a dvojkou. Jakmile je v *i* trojka, již nesouhlasí podmínka i < 3 a cyklus končí. O vynechávání složených závorek platí to samé, co u podmínek. V tomto případě tam nemusí být, protože cyklus spouští pouze jediný příkaz. Nyní můžeme místo trojky napsat do deklarace cyklu desítku. Příkaz se spustí 10x aniž bychom psali něco navíc. Určitě vidíte, že cykly jsou mocným nástrojem.

Zkusme si nyní využít toho, že se nám proměnná inkrementuje. Vypišme si čísla od jedné do deseti a za každým mezeru.

Spustit kód
Klikni pro editaci

```
for (int i = 1; i <= 10; i++)
{
System.out.printf("%d ", i);
}
```

Vidíme, že řídící proměnná má opravdu v každé iteraci (průběhu) jinou hodnotu.

Pokud vás zmátlo použití printf(), můžeme místo ní použít pouze print(), která na rozdíl od println() po vypsání neodřádkuje:

Spustit kód
Klikni pro editaci

```
for (int i = 1; i <= 10; i++)
{
System.out.print(i + " ");
}
```

Nyní si vypíšeme malou násobilku (násobky čísel 1 až 10, vždy do deseti). Stačí nám udělat cyklus od 1 do 10 a proměnnou vždy násobit daným číslem. Mohlo by to vypadat asi takto:

Spustit kód
Klikni pro editaci

```
for (int i = 1; i <= 10; i++)
{
System.out.print(i + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 2) + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 3) + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 4) + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 5) + " ");
}
System.out.println();
```

```
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 6) + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 7) + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 8) + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 9) + " ");
}
System.out.println();
for (int i = 1; i <= 10; i++)
{
System.out.print((i * 10) + " ");
}
```

```
Konzolová aplikace
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Program funguje hezky, ale pořád jsme toho dost napsali. Pokud vás napadlo, že v podstatě děláme 10x to samé a pouze zvyšujeme číslo, kterým násobíme, máte pravdu. Nic nám nebrání vložit 2 cykly do sebe:

Spustit kód

Klikni pro editaci

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

Poměrně zásadní rozdíl, že? Pochopitelně nemůžeme použít u obou cyklů *i*, protože jsou vložené do sebe. Proměnná *j*nabývá ve vnějším cyklu hodnoty 1 až 10. V každé iteraci (rozumějte průběhu) cyklu je poté spuštěn další cyklus s proměnnou *i*. Ten je nám již známý, vypíše násobky, v tomto případě násobíme proměnnou *j*. Po každém běhu vnitřního cyklu je třeba odřádkovat, to vykoná System.out.println().

Udělejme si ještě jeden program, na kterém si ukážeme práci s vnější proměnnou. Aplikace bude umět spočítat libovolnou mocninu libovolného čísla:

Spustit kód

Klikni pro editaci

```
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.println("Výsledek: " + vysledek);
System.out.println("Děkuji za použití mocninátoru");
```

Asi všichni tušíme, jak funguje mocnina. Pro jistotu připomenu, že například $2^3 = 2 * 2 * 2$. Tedy $a^n$ spočítáme tak, že n-1 krát vynásobíme číslo *a* číslem *a*. Výsledek si samozřejmě musíme ukládat do proměnné. Zpočátku bude mít hodnotu *a*a postupně se bude v cyklu pronásobovat. Pokud jste to nestihli, máme tu samozřejmě článek s algoritmem výpočtu libovolné mocniny .

Vidíme, že naše proměnná *vysledek* je v těle cyklu normálně přístupná. Pokud si však nějakou proměnnou založíme v těle cyklu, po skončení cyklu zanikne a již nebude přístupná.

```
Konzolová aplikace
Mocninátor
==========
Zadejte základ mocniny:
2
Zadejte exponent:
3
Výsledek: 8
Děkuji za použití mocninátoru
```

Už tušíme, k čemu se for cyklus využívá. Zapamatujme si, že je **počet opakování pevně daný**. Do proměnné cyklu bychom neměli nijak zasahovat ani dosazovat, program by se mohl tzv. zacyklit, zkusme si ještě poslední, odstrašující příklad:

Spustit kód
Klikni pro editaci

```
// tento kód je špatně
for (int i = 1; i <= 10; i++)
{
i = 1;
}
```

Au, 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.

While cyklus

While cyklus funguje jinak, jednoduše opakuje příkazy v bloku dokud platí podmínka. Syntaxe cyklu je následující:

```
while (podminka)
{
// příkazy
}
```

Pokud vás napadá, že lze přes while cyklus udělat i FOR cyklus, máte pravdu FOR je vlastně speciální případ while cyklu. While se ale používá na trochu jiné věci, často máme v jeho podmínce např. metodu vracející logickou hodnotu true/false. Původní příklad z for cyklu bychom udělali následovně pomocí while:

Spustit kód
Klikni pro editaci

```
int i = 1;
while (i <= 10)
{
System.out.print(i + " ");
i++;
}
```

To ale není ideální použití while cyklu. Vezmeme si naši kalkulačku z minulých lekcí a opět ji trochu vylepšíme, konkrétně o možnost zadat více příkladů. Program tedy hned neskončí, ale zeptá se uživatele, zda si přeje spočítat další příklad. Připomeňme si původní verzi kódu (je to ta verze se switchem, ale klidně použijte i tu bez něj, záleží na vás):

Spustit kód
Klikni pro editaci

```
System.out.println("Vítejte v kalkulačce");
System.out.println("Zadejte první číslo:");
float a = Float.parseFloat(sc.nextLine());
System.out.println("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.println("Výsledek: " + vysledek);
}
else
```

```
{
System.out.println("Neplatná volba");
}
System.out.println("Děkuji za použití kalkulačky.");
```

Nyní vložíme téměř celý kód do while cyklu. Naší podmínkou bude, že uživatel zadá "ano", budeme tedy kontrolovat obsah proměnné *pokracovat*. Zpočátku bude tato proměnná nastavena na "ano", aby se program vůbec spustil, poté do ní necháme načíst volbu uživatele:

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

System.out.println("Vítejte v kalkulačce");
String pokracovat = "ano";
while (pokracovat.equals("ano"))
{
System.out.println("Zadejte první číslo:");
float a = Float.parseFloat(sc.nextLine());
System.out.println("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.println("Výsledek: " + vysledek);
}
else
{
System.out.println("Neplatná volba");
}
System.out.println("Přejete si zadat další příklad? [ano/ne]");
pokracovat = sc.nextLine();
}
System.out.println("Děkuji za použití kalkulačky.");
```

Všimněte si, že **Stringy porovnáváme pomocí metody equals(), nikoli pomocí operátoru ==!** Je to dáno tím, že String je referenční datový typ. Podmínka ("Text" == "Text") je špatně, musíme psát ("Text".equals("Text")). V kapitole o objektově orientovaném programování pochopíme proč.

```
Konzolová aplikace
Vítejte v kalkulačce
Zadejte první číslo:
12
Zadejte druhé číslo:
128
Zvolte si operaci:
1 - sčítání
2 - odčítání
3 - násobení
4 - dělení
1
Výsledek: 140
Přejete si zadat další příklad? [ano/ne]
ano
Zadejte první číslo:
-10.5
Zadejte druhé číslo:
```

Naši aplikaci lze nyní používat vícekrát a je již téměř hotová. Příště si ukážeme práci s poli.

Již toho umíme docela dost, začíná to být zábava, že? 🙂

**7. díl - Pole v Javě**

Minule jsme si v našem Java seriálu ukázali cykly . Dnes si představíme datovou strukturu pole a vyzkoušíme si, co všechno umí.

Pole
Představte si, že si chcete uložit nějaké údaje o více prvcích. Např. chcete v paměti uchovávat 10 čísel, políčka šachovnice nebo jména 50ti uživatelů. Asi vám dojde, že v programování bude nějaká lepší cesta, než začít bušit proměnné uzivatel1, uzivatel2... až uzivatel50. Nehledě na to, že jich může být třeba 1000. A jak by se v tom potom hledalo? Brrr, takle ne

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.

| indexy | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 15 | 3 | 21 | 8 | 3 | 12 | 5 | 3 |

© itnetwork.cz

*(Na obrázku je vidět pole osmi čísel)*

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 dát velikost dle nějaké proměnné). Pole se muselo deklarovat s konstantní velikostí přímo ve zdrojovém kódu. Toto se obcházelo tzv. pointery a vlastními datovými strukturami, což často vedlo k chybám při manuální správě paměti a 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. Lze to samozřejmě obejít nebo použít jiné datové struktury, ale k tomu se dostaneme.

Možná vás napadá, proč se tu zabýváme s polem, když má evidentně mnoho omezení a existují lepší datové struktury. Odpověď je prostá: pole je totiž jednoduché. Nemyslím pro nás na pochopení (to také), ale zejména pro Javu. 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.

Pro hromadnou manipulaci s prvky pole se používají cykly.

Pole deklarujeme pomocí hranatých závorek:

```
int[] pole;
```

*Pole* je samozřejmě název naší proměnné. Nyní jsme však pouze deklarovali, že v proměnné bude pole intů. Nyní ho musíme založit, abychom ho mohli používat. Použijeme k tomu klíčové slovo **new**, které zatím nebudeme vysvětlovat. Spokojme se s tím, že je to kvůli tomu, že je pole referenční datový typ (můžeme chápat jako složitější typ):

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

Nyní máme v proměnné *pole* pole o velikosti deseti intů.

K prvkům pole potom přistupujeme přes hranatou závorku, pojďme na první index (tedy index 0) uložit číslo 1.

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

Plnit pole takhle ručně by bylo příliš pracné, použijeme cyklus a naplníme si pole čísly od 1 do 10. K naplnění použijeme for cyklus:

```
int[] pole = new int[10];
pole[0] = 1;
for (int i = 0; i < 10; i++) {
pole[i] = i + 1;
}
```

Abychom pole vypsali, můžeme za předchozí kód připsat:

Spustit kód
Klikni pro editaci

```
for (int i = 0; i < pole.length; i++) {
System.out.print(pole[i] + " ");
}
```

Všimněte si, že pole má konstantu length, kde je uložena jeho délka, tedy počet prvků. Stejně tak můžeme použít metodu size(), která vrátí stejný výsledek.

Konzolová aplikace
1 2 3 4 5 6 7 8 9 10

Můžeme použít zjednodušenou verzi cyklu pro práci s kolekcemi, známou jako foreach. Ten projede všechny prvky v poli a jeho délku si zjistí sám. Jeho syntaxe je následující:

```
for (datovytyp promenna : kolekce) {
// příkazy
}
```

Cyklus projede prvky v kolekci (to je obecný název pro struktury, které obsahují více prvků, u nás to bude pole) postupně od prvního do posledního. Prvek máme v každé iteraci cyklu uložený v dané proměnné.

Přepišme tedy náš dosavadní program pro foreach. Foreach **nemá řídící proměnnou**, není tedy vhodný pro vytvoření našeho pole a použijeme ho jen pro výpis.

Spustit kód
Klikni pro editaci

```
int[] pole = new int[10];
pole[0] = 1;
for (int i = 0; i < 10; i++) {
pole[i] = i + 1;
}
for (int i : pole) {
System.out.print(i + " ");
}
```

Výstup programu:
Konzolová aplikace
1 2 3 4 5 6 7 8 9 10

Pole samozřejmě můžeme naplnit ručně a to i bez toho, abychom dosazovali postupně do každého indexu. Použijeme k tomu složených závorek a prvky oddělujeme čárkou:

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

Pole často slouží k ukládání mezivýsledků, které se potom dále v programu používají. Když něco potřebujeme 10x, tak to nebudeme 10x počítat, ale spočítáme to jednou a uložíme do pole, odtud poté výsledek jen načteme.

Metody na třídě Arrays

Java nám poskytuje třídu Arrays, která obsahuje pomocné metody pro práci s poli.

K jejímu použití je třeba ji naimportovat:

```
import java.util.Arrays;
```

Pojďme se na ně podívat:

Sort()

Jak již název napovídá, metoda nám pole seřadí. Její jediný parametr je pole, které chceme seřadit. Je dokonce tak chytrá, že pracuje podle toho, co máme v poli uložené. Stringy třídí podle abecedy, čísla podle velikosti. Zkusme si seřadit a vypsat naši rodinku Simpsnů:

Spustit kód

Klikni pro editaci

```
String[] simpsonovi = {"Homer", "Marge", "Bart", "Lisa", "Meggie"};
Arrays.sort(simpsonovi);
for (String s : simpsonovi) {
System.out.print(s + " ");
}
```

Konzolová aplikace

Bart Homer Lisa Maggie Marge

Zkuste si udělat pole čísel a vyzkoušejte si, že to opravdu funguje i pro ně.

BinarySearch()

Když pole seřadíme, umožní nám v něm Java vyhledávat prvky. Metoda binarySearch() nám vrátí index prvního nalezeného prvku. V případě nenalezení prvku vrátí -1. Metoda bere dva parametry, prvním je pole, druhým hledaný prvek. Umožníme uživateli zadat jméno Simpsna a poté zkontrolujeme, zda je to opravdu Simpson. **Pole musí být opravdu setříděné, než metodu zavoláme!**

Spustit kód

Klikni pro editaci

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

String[] simpsonovi = {"Homer", "Marge", "Bart", "Lisa", "Meggie"};
System.out.println("Zadej Simpsna (z rodiny Simpsů): ");
String simpson = sc.nextLine();

Arrays.sort(simpsonovi);
int pozice = Arrays.binarySearch(simpsonovi, simpson);
if (pozice >= 0)
System.out.println("Jo, to je Simpson!");
else
System.out.println("Hele, tohle není Simpson!");
```

Konzolová aplikace

Zadej Simpsna (z rodiny Simpsů):

Homer

Jo, to je Simpson!

CopyOfRange()

copyOfRange() již podle názvu zkopíruje část pole do jiného pole. Prvním parametrem je zdrojové pole, druhým startovní pozice a třetím konečná pozice. Metoda vrací nové pole, které je výsekem původního pole.

Proměnná délka pole

Říkali jsme si, že délku pole můžeme definovat i za běhu programu, pojďme si to zkusit:

```
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);
```

Konzolová aplikace

Ahoj, spočítám ti průměr známek. Kolik známek zadáš?

5

Zadejte 1. číslo: 1

Zadejte 2. číslo: 2
Zadejte 3. číslo: 2
Zadejte 4. číslo: 3
Zadejte 5. číslo: 5
Průměr tvých známek je: 2.6

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.

U výpočtu průměru si všimněte, že při dělení je před jedním operandem napsáno (float), tím říkáme, že chceme dělit neceločíselně. Jistě si vzpomínáte, že při zadávání čísel při dělení jsme při 3 / 2 dostali výsledek 1 a při 3 / 2.0F dostali výsledek 1.5. Zde je princip stejný.

To by pro dnešek stačilo, můžete si s polem hrát. Příště na vás čeká překvapení

## 8. díl - Textové řetězce v Javě podruhé - práce s jednotlivými znaky

V milém dílu seriálu o Javě jsme se naučili pracovat s polem. Pokud jste vycítili nějakou podobnost mezi polem a textovým řetězcem, tak jste vycítili správně. Pro ostatní může být překvapením, že **String je v podstatě pole znaků (charů)** a můžeme s ním i takto pracovat. Pro přístup k jednotlivým znakům slouží metoda charAt(x), kde x udává index znaku v řetězci (počínaje 0).

Nejprve si vyzkoušejme, že to všechno funguje. Rozcvičíme se na jednoduchém vypsání znaku na dané pozici:

Spustit kód
Klikni pro editaci

```
String s = "Ahoj ITnetwork";
System.out.println(s);
System.out.println(s.charAt(2));
```

Výstup:

Konzolová aplikace

```
Ahoj ITnetwork
o
```

Zklamáním může být, že znaky na dané pozici jsou v Javě **read-only**, nemůžeme je tedy jednoduše změnit. Samozřejmě to jde udělat jinak, později si to ukážeme, zatím se budeme věnovat pouze čtení jednotlivých znaků.

Analýza výskytu znaků ve větě

Napišme si jednoduchý program, který nám analyzuje zadanou větu. Bude nás zajímat počet samohlásek, souhlásek a počet nepísmenných znaků (např. mezera nebo !).

Daný textový řetězec si nejprve v programu zadáme napevno, abychom ho nemuseli při každém spuštění psát. Až bude program hotový, nahradíme ho sc.nextLine(). Řetězec budeme projíždět cyklem po jednom znaku. Rovnou zde říkám, že neapelujeme na rychlost programu a budeme volit názorná a jednoduchá řešení.

Nejprve si připravme kód, definujme si samohlásky a souhlásky. Počet nepísmen nemusíme počítat, bude to délka řetězce mínus samohlásky a souhlásky. Abychom nemuseli řešit velikost písmen, celý řetězec na začátku převedeme na malá písmena. Připravme si proměnné, do kterých budeme ukládat jednotlivé počty. Protože se jedná o složitější kód, nebudeme zapomínat na komentáře.

*pozn.: Kdybyste věděli, jak se správně říká nepísmennému znaku, napište mi to prosím do komentáře pod článek*

```
// řetězec, který chceme analyzovat
String s = "Programátor se zasekne ve sprše, protože instrukce na šampónu byly: Namydlit, omýt, opakovat.";
System.out.println(s);
s = s.toLowerCase();

// inicializace počítadel
int pocetSamohlasek = 0;
int pocetSouhlasek = 0;

// definice typů znaků
String samohlasky = "aeiouyáéěíóúůý";
String souhlasky = "bcčdďfghjklmnpqrřsštťvwxzž";

// hlavní cyklus
for (char c : s.toCharArray()) {

}
```

Zpočátku si připravíme řetězec a převedeme ho na malá písmena. Počítadla vynulujeme. Na definice znaků nám postačí obyčejné Stringy. Hlavní cyklus nám projede jednotlivé znaky v řetězci *s*. Abychom mohli znaky iterovat (procházet cyklem), musíme si String převést na pole znaků. V úvodu jsem říkal, že String vlastně pole znaků je, ale ne plnohodnotné. Obsahuje něco navíc a něco mu chybí, např. možnost prvky iterovat cyklem. V cyklu tedy na *s* zavoláme metodu toCharArray(), která vrátí plnohodnotné pole znaků z řetězce *s*. V každé iteraci cyklu bude v proměnné *c*aktuální znak.

Pojďme plnit počítadla, pro jednoduchost již nebudu opisovat zbytek kódu a přesunu se jen k cyklu:

```
// hlavní cyklus
for (char c : s.toCharArray()) {
if (samohlasky.contains(String.valueOf(c))) {
pocetSamohlasek++;
}
else if (souhlasky.contains(String.valueOf(c))) {
pocetSouhlasek++;
}
}
```

Metodu contains() na řetězci již známe, jako parametr ji lze předat podřetězec. Bohužel nemůžeme předat znak char, musíme tedy znak převést na String. K tomu slouží výše uvedená metoda valueOf(). Daný znak *c* naší věty tedy nejprve zkusíme vyhledat v řetězce *samohlasky* a případně zvýšit jejich počítadlo. Pokud v samohláskách není, podíváme se do souhlásek a

případně opětovně zvýšíme jejich počítadlo. Nyní nám chybí již jen výpis na konec. V textu použijeme speciální sekvenci znaků "\n", ta způsobí odřádkování.

Spustit kód

Klikni pro editaci

```
System.out.printf("Samohlásek: %d\n", pocetSamohlasek);
System.out.printf("Souhlásek: %d\n", pocetSouhlasek);
System.out.printf("Nepísmenných znaků: %d\n", s.length() - (pocetSamohlasek + pocetSouhlasek));
```

```
Konzolová aplikace
Programátor se zasekne ve sprše, protože instrukce na šampónu byly: Namydlit, omýt, opakovat.
Samohlásek: 31
Souhlásek: 45
Nepísmenných znaků: 17
```

A je to!

ASCII hodnota

Možná jste již někdy slyšeli o ASCII tabulce. Zejména v éře operačního systému MS DOS prakticky nebyla jiná možnost, jak zaznamenávat text. Jednotlivé znaky byly uloženy jako čísla typu byte, tedy s rozsahem hodnot od 0 do 255. V systému byla uložena tzv. ASCII tabulka, která měla také 255 znaků a každému ASCII kódu (číselnému kódu) přiřazovala jeden znak.

Asi je vám jasné, proč tento způsob nepřetrval dodnes. Do tabulky se jednoduše nevešly všechny znaky všech národních abeced, nyní se používá unicode (UTF8) kódování, kde jsou znaky reprezentovány trochu jiným způsobem. Nicméně v Javě máme stále možnost pracovat s ASCII hodnotami jednotlivých znaků. Hlavní výhoda je 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 budou bohužel jen nějak rozházeny.

Zkusme si nyní převést znak do jeho ASCII hodnoty a naopak podle ASCII hodnoty daný znak vytvořit:

Spustit kód

Klikni pro editaci

```
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);
```

Převodům se říká přetypování, ale o tom se blíže pobavíme až později.

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. Algoritmus zde máme samozřejmě opět vysvětlený a to v článku Cézarova šifra . Program si dokonce můžete vyzkoušet v praxi - Online cézarova šifra .

Vraťme se k programování a připravme si kód. 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. Po dokončení nahradíme obsah proměnné metodou sc.nextLine(). Šifra nepočítá s diakritikou, mezerami a interpunkčními znaménky. Diakritiku budeme bojkovat a budeme předpokládat, že ji uživatel nebude zadávat. Ideálně bychom poté měli diakritiku před šifrováním odstranit, stejně tak cokoli kromě písmen.

```
// 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()) {

}

// 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:

Spustit kód

Klikni pro editaci

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

```
Konzolová aplikace
Původní zpráva: cernediryjsoutamkdebuhdelilnulou
Zašifrovaná zpráva: dfsofejszktpvubnlefcviefmjmovmpv
```

Program si vyzkoušíme. Výsledek vypadá docela dobře. Zkusme si však zadat vyšší posun nebo napsat slovo "zebra". Vidíme, že znaky mohou po "z" přetéct do ASCII hodnot dalších znaků, v textu tedy již nemáme jen písmena, ale další ošklivé znaky. Uzavřeme znaky do kruhu tak, aby posun plynule po "z" přešel opět k "a" a dále. Postačí nám k tomu jednoduchá podmínka, která od nové ASCII hodnoty odečne celou abecedu tak, abychom začínali opět na "a".

```java
int i = (int)c;
i += posun;
// kontrola přetečení
if (i > (int)'z') {
    i -= 26;
}
char znak = (char)i;
zprava += znak;
```

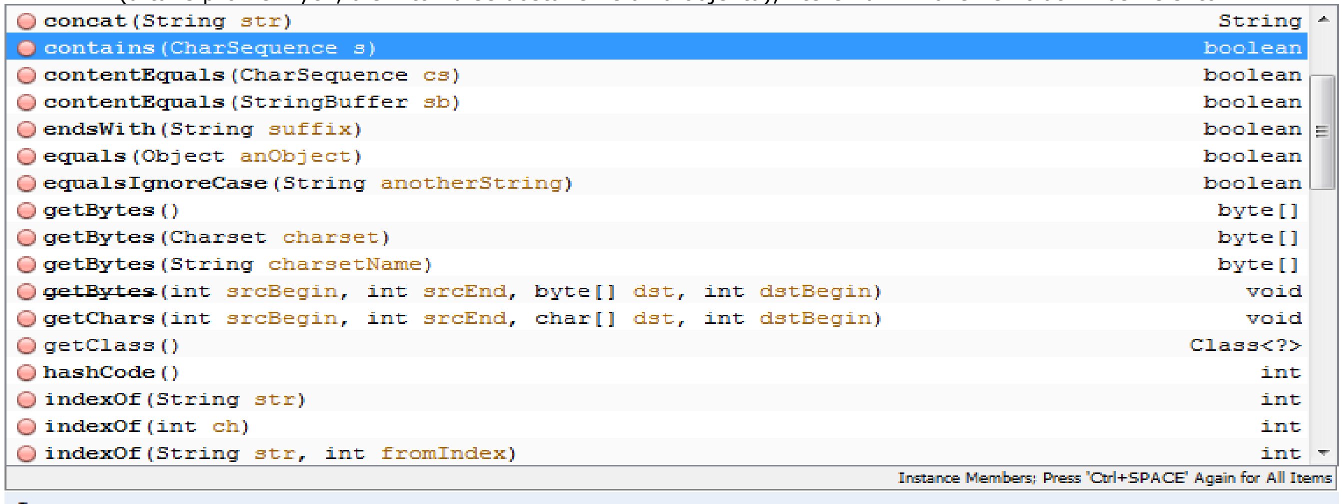
Pokud *i* přesáhne ASCII hodnotu 'z', snížíme ho o 26 znaků (tolik znaků má anglická abeceda). Operátor -= vykoná to samé, jako bychom napsali i = i - 26. Je to jednoduché a náš program je nyní funkční. Všimněme si, že nikde nepoužíváme přímé kódy znaků, v podmínce je (int)'z', i když bychom tam mohli napsat rovnou 122. Je to z důvodu, aby byl náš program plně odstíněn od explicitních ASCII hodnot a bylo lépe viditelné, jak funguje. Cvičně si zkuste udělat dešifrování.

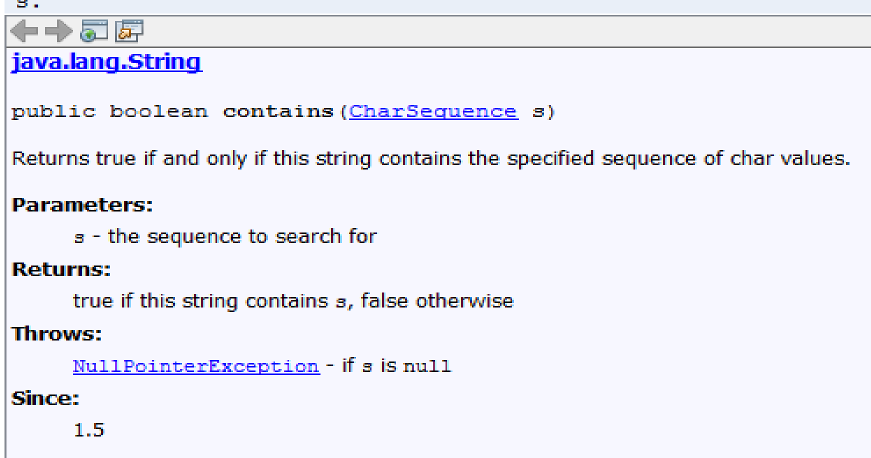
Příště si ukážeme, že String umí přeci jen ještě něco navíc. Prozradím, že budeme dekódovat Morzeovu abecedu.

**9. díl - Textové řetězce v Javě do třetice - Split**

Minule jsme si ukázali, že String je vlastně pole znaků. Dnes si vysvětlíme další metody na řetězci, které jsem vám záměrně zatajil, protože jsme nevěděli, že String je vlastně pole 🙂

Když si vytvoříme libovolnou proměnnou a napíšeme za ni poté tečku, zobrazí nám NetBeans nabídku všech metod a vlastností (a také proměnných, ale k tomu se dostaneme až u objektů), které na ni můžeme volat. Zkusme si to:




Tu samou nabídku lze vyvolat také stiskem CTRL + Mezerník v případě, že kurzor umístíme na tečku. Samozřejmě to platí pro všechny proměnné i třídy a budeme toho využívat stále častěji. Metody jsou řazené abecedně a můžeme jimi listovat pomocí kurzorových šipek. NetBeans nám zobrazuje popis metod (co dělají) a jaké vyžadují parametry.

Řekněme si o následujících metodách a ukažme si je na jednoduchých příkladech:

Další metody na řetězci

Substring()

Vrátí podřetězec od dané počáteční pozice do dané koncové pozice.

Spustit kód

Klikni pro editaci

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

Výstup:

Konzolová aplikace

naposled

CompareTo()

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

Spustit kód

Klikni pro editaci

```java
System.out.println("akát".compareTo("blýskavice"));
```

Výstup:

Konzolová aplikace

-1

Pojďme se nyní podívat na další metodu na Stringu, která je opravdu velmi užitečná.

Split()

Z předchozího tutoriálu víme, že parsování řetězce po znaku může být někdy docela složité a to jsme dělali poměrně jednoduchý příklad. S řetězci se samozřejmě budeme setkávat stále, a to jak na vstupu od uživatele (např. z konzole nebo z polí v okenních aplikacích), tak v souborech TXT a XML. Velmi často máme zadán jeden delší String (řádka souboru nebo řádka konzole), ve kterém je více hodnot, oddělených tzv. separátory, např. čárkou. V tomto případě hovoříme o formátu CSV (Comma-Separated Values, tedy hodnoty oddělené čárkou). Abychom si byli jisti, že víme, o čem hovoříme, ukažme si nějaké ukázkové řetězce:

```
Jan,Novák,Dlouhá 10,Praha 3,130 00
.. ... .-.. .- -. -.. ... --- ..-. -
(1,2,3;4,5,6;7,8,9)
```

První řetězec je očividně nějaký uživatel, takto bychom mohli např. realizovat uložení uživatelů do CSV souboru, každý na jeden řádek.

Druhý řetězec jsou znaky Morseovy abecedy, separátor (oddělovač) je zde mezera.

Třetí řetězec je matice o 3 sloupcích a 3 řádcích. Oddělovač sloupců je čárka, řádků středník

Na stringu můžeme volat metodu **split()**, která bere jako parametr separátor. Následně původní řetězec rozdělí podle separátorů na pole podřetězců, které vrátí. To nám velmi ulehčí práci při rozdělování hodnot v řetězci.

Jelikož neumíme tvořit objekty (uživatele) a ani pracovat s vícerozměrnými poli (matice), zkusíme si naprogramovat dekodér zpráv z Morseovy abecedy.

Dekodér Morseovy abecedy

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 morzeovce. 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:

```java
// 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 podřetězců, obsahujících jednotlivé znaky morzeovky. Splitovat budeme podle znaku mezery. Pole následně proiterujeme cyklem foreach:

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

// iterace znaků morzeovky
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 morzeovky 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 jak pole tak řetězec obsahují stejné znaky, seřazené dle abecedy. Umístěme do těla cyklu následující kód:

```java
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 (morzeův 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:

Spustit kód

Klikni pro editaci

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

Konzolová aplikace
Původní zpráva: .. - -. . - .-- --- .-. -.-
Dekódovaná zpráva: itnetwork

Hotovo! Za úkol máte si naprogramovat program opačný, který naopak zakóduje řetězec do morzeovky, kód bude velmi podobný. Se split() se potkáme během seriálu ještě několikrát.

Speciální znaky a escapování

Textový 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:

Spustit kód
Klikni pro editaci

```java
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, když chceme napsat samotné "\", musíme ho tzv. odescapovat:

Spustit kód
Klikni pro editaci

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

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

Spustit kód
Klikni pro editaci

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

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

Tímto jsme v podstatě zakončili sekci se základní strukturou jazyka Java, příště si uvedeme bonusový díl o matematické třídě. Ze základních konstrukcí jazyka vás tu ale již nic nepřekvapí 🙂 V podstatě byste již klidně mohli jít i na objekty, doporučují ale zbylé články ještě alespoň projet, jedná se přeci jen stále o základní znalosti, které byste měli mít.

## 10. díl - Vícerozměrná pole v Javě

V minulém tutoriálu o základech Javy jsme si uvedli metodu split() na textových řetězcích. Dnešní díl je v sekci základních konstrukcí Javy v podstatě bonusový a pojednává o tzv. vícerozměrných polích. Teoreticky můžete rovnou přejít k objektově orientovanému programování, doporučuji však si konec této sekce ještě alespoň projít, abyste měli o zbývajících technikách povědomí, přeci jen se jedná o dosti základní vědomosti.

Již umíme pracovat s jednorozměrným polem, které si můžeme představit jako řádku přihrádek v paměti počítače.

| indexy | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 15 | 3 | 21 | 8 | 3 | 12 | 5 | 3 |

© itnetwork.cz

*(Na obrázku je vidět pole osmi čísel)*

Ačkoli to není tak časté, v programování se občas setkáváme i s vícerozměrnými poli a to zejména pokud programujeme nějakou simulaci (např. hru).

Dvourozměrné pole

Dvourozměrné pole si můžeme v paměti představit jako tabulku a mohli bychom takto reprezentovat např. rozehranou partii piškvorek. Pokud bychom se chtěli držet reálných aplikací, které budete později v zaměstnání tvořit, můžeme si představit, že do 2D pole budeme ukládat informace o obsazenostech sedadel v kinosálu. Situaci bychom si mohli graficky znázornit např. takto:

| Y: \ X: | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 |

© itnetwork.cz

*(Na obrázku je vidět 2d pole reprezentující obsazenost kinosálu)*

Kinosál by byl v praxi samozřejmě větší, ale jako ukázka nám toto pole postačí. 0 znamená volno, 1 obsazeno. Později bychom mohli doplnit i 2 - rezervováno a podobně. Pro tyto stavy by bylo správnější vytvořit si vlastní datový typ, tzv. výčet, ale s ním se setkáme až později, takže si teď musíme vystačit pouze s čísly.

Java ve skutečnosti neposkytuje žádnou dodatečnou podporu pro vícerozměrná pole, můžeme si je však jednoduše deklarovat jako pole polí. Definice takového pole pro kinosál by vypadala takto:

```java
int[][] kinosal = new int[5][5];
```

První číslice udává počet sloupců, druhá počet řádků (samozřejmě si to můžeme určit i obráceně, např. matice v matematice se zapisují opačně).

Všechna číselná pole v Javě jsou po deklaraci automaticky inicializována samými nulami, můžeme se na to spolehnout. Vytvořili jsme si tedy v paměti tabulku plnou nul.

Naplnění daty

Nyní kinosál naplníme jedničkami tak, jak je vidět na obrázku výše. Protože budeme jako správní programátoři líní, využijeme k vytvoření řádku jedniček for cykly 🙂 Pro přístup k prvku 2D pole musíme samozřejmě zadat 2 souřadnice.

```java
kinosal[2][2] = 1; // Prostředek
for (int i = 1; i < 4; i++) // 4. řádek
{
kinosal[i][3] = 1;
}
for (int i = 0; i < 5; i++) // Poslední řádek
```

```
{
kinosal[i][4] = 1;
}
```

Výpis

Výpis pole opět provedeme pomocí cyklu, na 2d pole budeme potřebovat cykly 2 (jeden nám proiteruje sloupce a druhý řádky). Jako správní programátoři nevložíme počet řádků a sloupců do cyklů napevno, jelikož se může změnit. Musíme však pamatovat na skutečnost, že když se zeptáme na kinosal.length, bude obsahovat počet sloupců (přesněji délku vnějšího pole, které představuje sloupce). Abychom získali počet řádků (délku vnitřního pole, které sloupec reprezentuje), zeptáme se na kinosal[0].length. Všimněte si, že na to v poli musí být tedy alespoň jeden řádek.
Cykly zanoříme do sebe tak, aby nám vnější cyklus projížděl řádky a vnitřní sloupce v aktuálním řádku. Po výpisu řádku je nutné odřádkovat. Oba cykly musí mít samozřejmě jinou řídící proměnnou:

Spustit kód

Klikni pro editaci

```
for (int j = 0; j < kinosal[0].length; j++)
{
for (int i = 0; i < kinosal.length; i++)
{
System.out.print(kinosal[i][j]);
}
System.out.println();
}
```

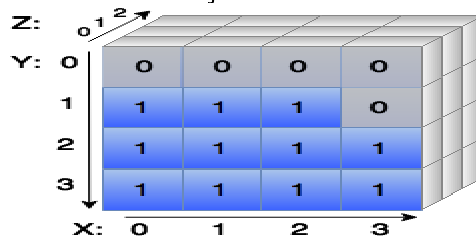
Výsledek:

Konzolová aplikace

```
00000
00000
00100
01110
11111
```

N-rozměrná pole

Někdy může být příhodné vytvořit si pole o ještě více dimenzích. My všichni si jistě dokážeme představit minimálně 3D pole. S příkladem s kinosálem se nabízí případ užití, kdy má budova více pater (nebo obecně více kinosálů). Vizualizace by vypadala asi nějak takto:




3D pole můžeme vytvořit tím samým způsobem, jako 2D pole:

```
int[][][] kinosaly = new int [5][5][3];
```

Kód výše vytvoří 3D pole jako na obrázku. Přistupovat k němu budeme opět přes indexery (hranaté závorky) jako předtím, jen již musíme zadat 3 souřadnice.

```
kinosaly[3][2][1] = 1; // Druhý kinosál, třetí řada, čtvrtý sloupec
```

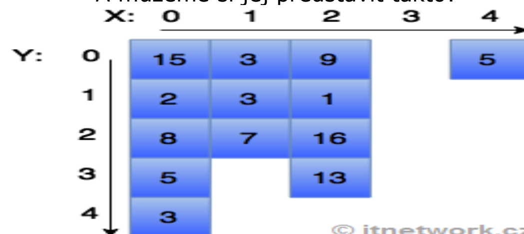
Pokud se zeptáme na kinosaly[0][0].length, získáme počet "pater" (kinosálů).

Zubatá pole

Pro všechny sloupce nemusíme udávat tu samou délku. Výsledkem je potom pole "zubaté" (v anglické literatuře jako "jagged array"). Výhodou deklarování 2D polí tímto způsobem je, že do každého řádku/sloupce můžeme poté uložit jak velké pole chceme. Tímto způsobem můžeme ušetřit paměť.
Takové 2D pole deklarujeme následujícím způsobem:

```
int[][] kinosal = new int[5][];
```

A můžeme si jej představit takto:




Nevýhodou tohoto přístupu je, že musíme pole nepříjemně inicializovat sami. Původní řádek s pěti buňkami sice existuje, ale jednotlivé sloupečky si do něj musíme navkládat sami (zatím si vložme všechny sloupečky o 5ti prvcích):

```
for (int i = 0; i < kinosal.length; i++)
{
kinosal[i] = new int[5];
}
```

Java rovněž dále neposkytuje žádný komfort ve formě získání počtu sloupců a řádků polí polí. Velikost pole musíme získat takto:

```
int sloupcu = kinosal.length;
int radku = 0;
if (sloupcu != 0)
radku = kinosal[0].length;
```

Všimněte si, že je nutné ptát se na počet sloupců, pokud je totiž 0, nemůžeme se dostat k 1. sloupci, abychom zjistili jeho délku (počet řádků ve sloupci).

K hodnotám v poli poté přistupujeme pomocí 2 indexerů:

```
kinosal[4][2] = 1; // Obsazujeme sedadlo v 5. sloupci a 3. řadě
```

Zkrácená inicializace vícerozměrných polí

Ještě si ukážeme, že i vícerozměrná pole je možné rovnou inicializovat hodnotami (kód vytvoří rovnou zaplněný kinosál jako na obrázku):

```
int[][] kinosal = {
{ 0, 0, 0, 0, 1 },
{ 0, 0, 0, 1, 1 },
{ 0, 0, 1, 1, 1 },
{ 0, 0, 0, 1, 1 },
{ 0, 0, 0, 0, 1 }
};
```

(Pole je v tomto zápisu otočené, jelikož definujeme sloupce, které zde zapisujeme jako řádky).

Podobnou inicializaci můžeme použít dokonce i u polí zubatých (kód níže vytvoří zubaté pole jako na obrázku):

```
int[][] zubatePole = {
new int[] {15, 2, 8, 5, 3},
new int[] {3, 3, 7},
new int[] {9, 1, 16, 13},
new int[] {},
new int[] {5}
};
```

Na závěr bych rád dodal, že někteří lidé, kteří neumí správně používat objekty, využívají 2D polí k ukládání více údajů o jediné entitě. Např. budeme chtít uložit výšku, šířku a délku pěti mobilních telefonů. Ačkoli se vám nyní může zdát, že se jedná o úlohu na 3D pole, ve skutečnosti se jedná o úlohu na obyčejné 1D pole (přesněji seznam) objektů typu Telefon. Ale o tom až u objektově orientovaného programování. Pole si můžete vyzkoušet ještě v cvičení v této sekci.

Příště se podíváme na matematické funkce a základní seriál zakončíme.

## 11. díl - Matematické funkce v Java a knihovna Math

Náš seriál o Javě teď vlastně teprve začíná, nicméně v této sekci s tutoriály o těch nejzákladnějších konstrukcích jazyka jsme již u konce. Jsem rád, že jsme se úspěšně dostali až sem, další sekce se totiž bude věnovat objektově orientovanému programování. Budeme tam vytvářet opravdu zajímavé aplikace a i jednu hru. Sekci zakončeme odlehčujícím článkem s přehledem matematických funkcí, které se nám v našich programech jistě budou v budoucnu hodit.

Základní matematické funkce jsou v Javě obsaženy v **třídě Math**. Třída nám poskytuje dvě základní konstanty: PI a E. PI je pochopitělně číslo Pí (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:

Spustit kód

Klikni pro editaci

```
System.out.println("Pí: " + Math.PI);
System.out.println("e: " + 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.

Konzolová aplikace

Pí: 3.141593

e: 2.718282

Pojďme si nyní popsat metody, které třída poskytuje:

### Metody na třídě Math

#### min(), max()

Začněme s tím jednodušším 🙂 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:

Spustit kód

Klikni pro editaci

```
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:

Spustit kód

Klikni pro editaci

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

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 $8^0 = 1$.

Dělení

Programovací jazyky se často odlišují tím, jak v nich funguje dělení čísel. Tuto problematiku je nutné dobře znát, abyste nebyli poté (nepříjemně) překvapeni. Napišme si jednoduchý program:

Spustit kód

Klikni pro editaci

```
int a = 5 / 2;
double b = 5 / 2;
double c = 5.0 / 2;
double d = 5 / 2.0;
double e = 5.0 / 2.0;
// int f = 5 / 2.0;

System.out.println(a);
System.out.println(b);
System.out.println(c);
System.out.println(d);
System.out.println(e);
```

V kódu několikrát dělíme 5 / 2, což je matematicky 2.5. Jistě ale tušíte, že výsledek nebude ve všech případech stejný. Troufnete si tipnout si co kdy vyjde? Zkuste to

Kód by se nepřeložil kvůli řádku s proměnnou f, proto jsme ho zakomentovali. Problém je v tom, že v tomto případě vyjde desetinné číslo, které se snažíme uložit do čísla celého (int). Výstup programu je poté následující:

Konzolová aplikace

```
2
2.0
2.5
2.5
2.5
```

Vidíme, že výsledek dělení je někdy celočíselný a někdy reálný. Přitom vůbec nezáleží na datovém typu proměnné, do které výsledek ukládáme, ale na datovém typu čísel, které dělíme. Pokud je jedno z čísel desetinné, je výsledek vždy desetinné číslo. 2 celá čísla vrátí vždy zas celé číslo, dejte si na to pozor např. když budete počítat průměr, pro desetinný výsledek je nutné alespoň jednu proměnnou přetypovat na desetinné číslo.

```
int soucet = 10;
int pocet = 4;
double prumer = (double)soucet / pocet;
```

*Pozn.: Např. v jazyce PHP je výsledek dělení vždy desetinný, až budete dělit v jiném programovacím jazyce než v Javě, zjistěte si jak dělení funguje než jej použijete.*

Zbytek po celočíselném dělení

V našich aplikacích můžeme často potřebovat zbytek po celočíselném dělení (tzv. modulo). U našeho příkladu 5 / 2 je celočíselný výsledek 2 a modulo 1 (zbytek). Modulo se často používá pro zjištění zda je číslo sudé (zbytek po dělení 2 je 0), když chcete např. vybarvit šachovnici, zjistit odchylku vaší pozice od nějaké čtvercové sítě a podobně.

V Javě a obecně v céčkových jazycích zapíšeme modulo jako %:

Spustit kód

Klikni pro editaci

```
System.out.println(5 % 2); // Vypíše 1
```

Tak to bychom měli. V sekci [Základní konstrukce jazyka Java] zájemci naleznou ještě několik dalších článků a příkladů k procvičení. Seriál nyní pokračuje v sekci [Základy objektově orientovaného programování v Javě] . [Příště] si tedy představíme objektový svět a pochopíme mnoho věcí, které nám až doteď byly utajovány

Programujeme jednoduchou hru v Javě: Šibenice

Připravil jsem si pro vás tutoriál, ve kterém se naučíme vytvořit všem známou hru "Šibenice" v konzoli. K tomu budeme potřebovat jen [základní konstrukce jazyka Java] a pár maličkostí, které budou v průběhu tutoriálu vysvětleny.

Budeme potřebovat vědět, kolikrát uživatel zadal špatné písmeno. K tomu nám poslouží proměnná *trest*, která bude počítat "trestné body". Dále budeme potřebovat proměnnou *vyhra*, která nám naopak bude počítat kolik správných písmen uživatel uhodl. Hra by šla vytvořit i bez této proměnné, ale pro jednoduchost ji tu necháme. Bez čeho se ale neobejdeme bude hledané slovo, které bude ukryto v proměnné *slovo*. V poslední řadě potřebujeme proměnnou, která si bude pamatovat postup hry, a to bude *postup*.

```
int trest = 0;
int vyhra = 0;
String slovo = "jablko";
char[] postup = new char[slovo.length()];
```

Nyní, když máme proměnné vytvořené, vložíme do postup tolik pomlček, kolik je písmen ve slově. K tomu nám poslouží jednoduchý cyklus.

```
for(int i=0;i!=slovo.length();i++)
postup[i] = '-';
```

Máme vše potřebné připravené, tak se můžeme pustit do hlavního cyklu, který bude řídit celou aplikaci. Vytvoříme ho pomocí cyklu while, který se opakuje tak dlouho, dokud platí podmínka. V našem případě se ptáme, jestli uživatel nedosáhl hranici 7 trestných bodů a zároveň neuhodl všechny písmena.

```
while (trest!=7&&vyhra!=slovo.length())
{
```

```
...
}
```

Hned na začátek vypíšeme *postup*, ať uživatel ví, kolikatipísmenné slovo má uhodnout. Následně ho vyzveme, aby zadal písmeno, které si myslí, že je obsaženo ve slově. Pro případ, že by uživatel zadal velké písmeno, použijeme metodu toLowerCase(). Poté si přes podmínku ověříme, zda je zadané písmeno obsaženo ve slově. Pokud ano, projedeme si jednotlivá písmena a hledáme, která se shodují. Ty následně předáme na *postup* a přičteme 1 do *vyhra*. Je možné, že uživatel zadá opakovaně jedno písmeno, proto podmínku rozšíříme o podmínku, zda-li *postup* již toto písmeno neobsahuje. Pokud není zadané písmeno obsaženo ve slově, přičteme 1 do *trest*.

```
for(char c:postup)
System.out.printf("%c",c);
System.out.printf("\nZadejte písmeno: ");
String volba = sc.nextLine();
volba = volba.toLowerCase();

if(slovo.contains(volba))
{
for(int i=0;i!=slovo.length();i++)
if(volba.equals(Character.toString(slovo.charAt(i)))&&
postup[i]!=slovo.charAt(i))
{
postup[i]=slovo.charAt(i);
vyhra+=1;
}
}
else
trest+=1;
```

Už se nám sčítají trestné body, můžeme tedy začít vykreslovat šibenici, a to pomocí switch. Záleží na vás, jak bude vypadat, já jsem si zvolil následující.

```
switch (trest)
{
case 7:
System.out.println("     ____________\n"
+ "    ||       I\n"
+ "    ||      ( )\n"
+ "    ||       |\n"
+ "    ||      /|\\\n"
+ "    ||       |\n"
+ "    ||      / \\\n"
+ "   .---.\n"
+ "  /    \\");
System.out.println("Prohrál jste!\n"
+ "Hledané slovo bylo: " + slovo);
break;
case 6:
System.out.println("     ____________\n"
+ "    ||       I\n"
+ "    ||      ( )\n"
+ "    ||       |\n"
+ "    ||       |\n"
+ "    ||       |\n"
+ "    ||      / \\\n"
+ "   .---.\n"
+ "  /    \\");
break;
case 5:
System.out.println("     ____________\n"
+ "    ||       I\n"
+ "    ||      ( )\n"
+ "    ||       |\n"
+ "    ||       |\n"
+ "    ||       |\n"
+ "    ||\n"
+ "   .---.\n"
+ "  /    \\");
break;
case 4:
System.out.println("     ____________\n"
+ "    ||       I\n"
+ "    ||      ( )\n"
+ "    ||\n"
+ "    ||\n"
+ "    ||\n"
+ "    ||\n"
+ "   .---.\n"
+ "  /    \\");
break;
```

```
case 3:
System.out.println("     ____________\n"
        + "    ||       I\n"
        + "    ||\n"
        + "    ||\n"
        + "    ||\n"
        + "    ||\n"
        + "    ||\n"
        + "   .---.\n"
        + "  /     \\");
break;
case 2:
System.out.println("\n    ||\n"
        + "    ||\n"
        + "    ||\n"
        + "    ||\n"
        + "    ||\n"
        + "    ||\n"
        + "   .---.\n"
        + "  /     \\");
break;
case 1:
System.out.println("\n\n\n\n\n\n\n   .---.\n"
        + "  /     \\");
break;
default :
}
```

Jelikož nám cyklus skončí ve dvou případech (výhra, prohra), musíme zjistit, proč skončil a vypsat vítězství nebo porážku. Prohra je jednoduchá, tu připíšeme k poslední fázi stavby šibenice. Výpis výhry dáme do podmínky, kterou umístíme za cyklus.

```
if (trest != 7)
System.out.println("Vyhrál jste!");
```

Šibenice nám krásně funguje, ale stále se tam objevuje jedno slovo. Aby byla hra zábavnější, vytvoříme si na začátku pole slov, které se mohou objevit a vzápětí náhodný výběr z těchto slov. Náhodné číslo nám zajistí metoda Math.random(), která vrací desetinné číslo od 0 do 1. My bychom potřebovali číslo od 0 do počtu prvků v poli slov -1 (protože jsou číslovány od 0). Proto náhodné číslo vynásobíme délkou pole a přetypujeme na int.

```
String[] databaze = {"auto", "pocitac", "internet", "skola"};
String slovo = databaze[(int)(Math.random()*databaze.length)];
```

Teď je již šibenice kompletní.

```
Output - Sibenice (run)
      ____________
      | |         I
      | |
      | |
      | |
      | |
      | |
     .---.
    /     \
sk--a
Zadejte písmeno: o
      ____________
      | |         I
      | |
      | |
      | |
      | |
      | |
     .---.
    /     \
sko-a
Zadejte písmeno: l
```

Do databaze si můžete napsat slov kolik chcete. Pokud jste něčemu nerozuměli nebo něco vám nefunguje, můžete si aplikaci pod článkem stáhnout.

Programujeme jednoduchou hru v Javě: Logik

V dnešním tutoriálu si ukážeme jak naprogramovat hru Logik jen se znalostí základních konstrukcí jazyka Java.

Pokud hru někdo nezná - Jedná se o logickou hru, kde je cílem uhodnout čtyřmístné číslo (ve skutečnosti se hádají barvy, ale pro jednoduchost budeme brát v úvahu jen čísla).

Pro začátek by bylo vhodné si definovat proměnnou, která v sobě bude obsahovat kolik pozic bude mít hledaná kombinace čísel, to bude proměnná *pocetCisel*. Na několika místech se totiž budeme na tuto proměnou odkazovat, navíc bude jednoduché hru "oživit" přidáním dalších pozic.

```
int pocetCisel = 4;
```

Nyní víme, jak má být číslo velké, proto si ho již můžeme vytvořit. Pro přesnost použijeme pole čísel *cislo*. Použijeme k tomu for cyklus s metodou Math.random(), kterou vynásobíme 10, aby se objevovaly cifry 0 až 9, pokud chcete hru trochu ulehčit, můžete metodu vynásobit menším číslem. Metoda nám vrací double, proto ji přetypujeme na int.

```
int cislo[] = new int[pocetCisel];
for(int i=0;i<pocetCisel;i++)
cislo[i]=(int)(Math.random()*10);
```

Máme připravené hledané číslo, nyní si přichystáme pomocné proměnné. Budou to *vstup*, do které budeme načítat pokus uživatele, *presne*, která bude reprezentovat počet cifer, které jsou správné a správně umístěné, dále pak *priblizne*, která bude uchovávat počet cifer, které jsou s porovnáním se vstupem v proměnné *cislo*, ale na špatné pozici. Nakonec si jen přidáme instanci Scanneru.

```
String volba;
int presne = 0;
```

```
int priblizne = 0;
Scanner sc = new Scanner(System.in,"UTF-8");
```

Řídící cyklus

Řídící cyklus bude ovládat celou naši hru. Jelikož je to logická hra a žádná typovačka, použijeme for cyklus s 10 průběhy pro 10 pokusů.

```
for(int a=0; a<10; a++)
{
...
}
```

Na začátku každého průběhu se uživatele zeptáme na kombinaci, kterou následně načteme do *volba*. Pro případ špatného vstupu uživatele přidáme na konec pár mezer. Dále vynulujeme proměnné *presne* a *priblizne*.

```
System.out.printf("Zadejte další kombinaci: ");
volba = sc.nextLine()+ "      ";
presne = priblizne = 0;
```

Už známe typovací kombinaci, můžeme tedy začít porovnávat. Jelikož potřebujeme porovnat každou cifru, použijeme opět for cyklus s počtem opakování roven *pocetCisel*.

```
for(int i=0; i<pocetCisel; i++)
{
...
}
```

Do něj vložíme podmínku, jestli cifra v proměnné *číslo* na *i*. pozici je rovna cifře na *i*. pozici vstupu. Pokud ano, přičteme do *presne* jedničku. Pokud ne, pokračujeme v else.

```
if (cislo[i] == volba.charAt(i) - 48)
presne++;
else
{
...
}
```

Zde budeme porovnávat každou cifru v *cislo* s každou cifrou ve *vstup.* Proto potřebujeme for cyklus opět s počtem opakování roven *pocetCisel*. V něm bude podobná podmínka, jaká byla při ověřování přesnosti. Zde ale budeme porovnávat v *cislo* cifru na *i*. pozici a ve *vstup* na *j*. pozici. Platí-li podmínka, přičteme jedničku do *priblizne* a ukončíme cyklus. Docílíme toho tím, že *j* bude vyšší číslo, než jaké je uvedeno v podmínce cyklu.

```
for(int j=0;j<pocetCisel;j++)
{
if(cislo[i]==volba.charAt(j)-48)
{
priblizne++;
j = 10;
}
}
```

Nyní vyskočíme zpět do řídícího cyklu a přidáme podmínku pro výhru. Přesněji pokud *presne* je rovno *pocetCisel*. Pokud podmínka platí, vypíšeme zprávu s výhrou a ukončíme cyklus podobně jako před chvílí. Jinak vypíšeme stav aktuální kombinace ku hledanému číslu.

```
if (presne == pocetCisel)
{
System.out.println("Gratuluji, vyhrál jste!");
a = 10;
}
else
System.out.printf("Správně: %d, přibližně: %d\n\n",presne, priblizne);
```

Nakonec našeho programu, až za řídící cyklus, vložíme podobnou podmínku s tím rozdílem, že bude negovaná. Bude sloužit pro vypsání prohry. Samozřejmě tam přidáme i výpis hledaného čísla a to pomocí for each cyklu.

```
if(presne!=pocetCisel)
{
System.out.printf("Prohra! Hledaná kombinace byla: ");
for (int i:cislo)
System.out.printf("%d", i);
}
```

Nyní máme všechno hotové a můžeme si ji vyzkoušet. Určitě ji můžete všelijak upravit → změnit počet pokusů, změnit počet cifer hledané kombinace nebo zmenšit číslo, které se může objevovat v kombinaci.

```
Output - Logik (run)
Zadejte další kombinaci: 3211
Správně: 0, přibližně: 1

Zadejte další kombinaci: 8456
Správně: 0, přibližně: 3

Zadejte další kombinaci: 4568
Správně: 0, přibližně: 3

Zadejte další kombinaci: 5684
Správně: 3, přibližně: 0

Zadejte další kombinaci: 5682
Správně: 2, přibližně: 0

Zadejte další kombinaci:
INS
```

Pokud jste něčemu nerozuměli nebo něco vám nefunguje, můžete si aplikaci pod článkem stáhnout.