Java pro začátečníky (11) - Rekurze

V tomto dílu se budeme zabývat dalším způsobem, kterak vytvořit v Javě cyklus – rekurzí. Ukážeme si rekurzivní algoritmus pro výpočet faktoriálu. Řekneme si, jaké jsou výhody a nevýhody takto formulovaných algoritmů. Vzpomeneme si na Fibonacciho posloupnost (která je z definice rekurzivní), ale dnes nám poslouží spíše jako odstrašující příklad. V úplném závěru zmíníme nástroj, který výrazně usnadní vyhledávání chyb v našich programech – *debugger*.

Co to je rekurze?

Rekurze je definování nějakého objektu (chápáno matematicky) pomocí sebe sama (rekurzivní zkratka: *GNU – GNU's Not Unix!*). Rekurzivní funkce je taková funkce, která volá sama sebe. Pokud volá sama sebe přímo, tak rekurzi označíme za *přímou*, pokud skrze nějakého prostředníka, tak za *nepřímou*.

Rekurzivní funkce musí být na tento typ volání připravena. Musí obsahovat zarážku – triviální případ – pro který je již přímo definován výsledek. Tím je zajištěno, že se funkce nebude volat donekonečna, čímž by velmi rychle došla paměť na zásobníku a program by havaroval (volání funkcí probíhá interně pomocí zásobníkové struktury).

Příklad

Funkce faktoriál je asi nejjednodušším případem rekurzivní funkce, protože výsledek volání $fac(n)$ je roven součinu $n \cdot fac(n-1)$, tj.:

$$fac(n) = n \cdot fac(n-1),\ fac(0) = 1,\ fac(1) = 1$$

Výpočet faktoriálu čísla 4:

$$fac(1) = 1$$
$$fac(2) = 2 \cdot fac(1) = 2$$
$$fac(3) = 3 \cdot fac(2) = 6$$
$$fac(4) = 4 \cdot fac(3) = 24$$

Pokud bychom si v hlavě simulovali rekurzi, tak bychom začali na posledním řádku postupu a ptali bychom se, kolik je $4!$. Tuto odpověď sice neznáme, ale víme, že k ní potřebujeme odpověď na to, kolik je $3!$. Ani to ale nevíme, musíme sestoupit na $2!$, což také nevíme. Při dalším sestupu ale již narážíme na triviální případ $1! = 1$ a výsledek můžeme vrátit vzhůru. Podle předpisu nejprve získáme výsledek $2!$, pak $3!$ a nakonec i $4!$, což byla původní otázka.

Java

```
01. /**
02.  * Vypocita rekurzivne faktorial cisla
03.  * @param number cislo >=0
04.  * @return faktorial cisla, -1 v pripade neplatneho vstupu
```

```java
 */
public static int factorialRek(int number){
    if(number < 0) return -1;
    if(number == 0 || number == 1) return 1;
    return number * factorialRek(number - 1);
}
```

K čemu je to dobré?

Nyní každého napadne, že je to poněkud složité a těžkopádné. Ano, pro výpočet faktoriálu zajisté. Ale existuje řada postupů a funkcí, které jsou rekurzivní z definice (vzpomeňme si na Fibonacciho posloupnost). Dále je zde řada datových struktur (graf, strom...), které jsou buď rekurzivní samy o sobě nebo je práce nad nimi pomocí rekurze jednodušší. V neposlední řadě zde jsou algoritmy *divide-et-impera* (*rozděl a panuj*), které jsou na štěpení problému na malé snadno řešitelné díly přímo založené (zkuste si přečíst články například o *Merge sortu* a *Quicksortu* pokud nepochopíte vše, nevadí).

V kostce řečeno: mnohdy je rekurzivní formulace výrazně jednodušší.

Nevýhody rekurze

Než přistoupíme k odstrašujícímu příkladu, tak si řekneme některé dílčí nevýhody. Tou první je samotné volání funce, které s sebou nese určitý overhead a tím i dopad na rychlost aplikace. Dobrou zprávou je, že toto zpomalení není nijak zásadní.

Druhou výraznější nevýhodou je nižší flexibilita takto formulovaných algoritmů. Některé pokročilejší algoritmy jsou ve své rekurzivní podobě naprosto odlišné, zatímco v linearizované podobě (rekurze je nahrazena explicitním použitím struktury *zásobníku*) jsou skoro totožné.

Poslední nevýhoda je asi nejvýraznější – pouze velmi obtížně lze odhadout (a spočítat) *asymptotickou složitost* daného algoritmu.

Fibonacciho posloupnost

Na Fibonacciho posloupnost jsme již narazili v osmém dílu. Řekli jsme si, že ji matematicky můžeme zapsat jako:

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1$$

Což je ve světle dnešních znalostí rekurze. Abychom byli úplně přesní, jedná se o *stromovou rekurzi*, protože je daná funkce v každém kroku volána více jak jedenkrát a program se proto větví.

Java

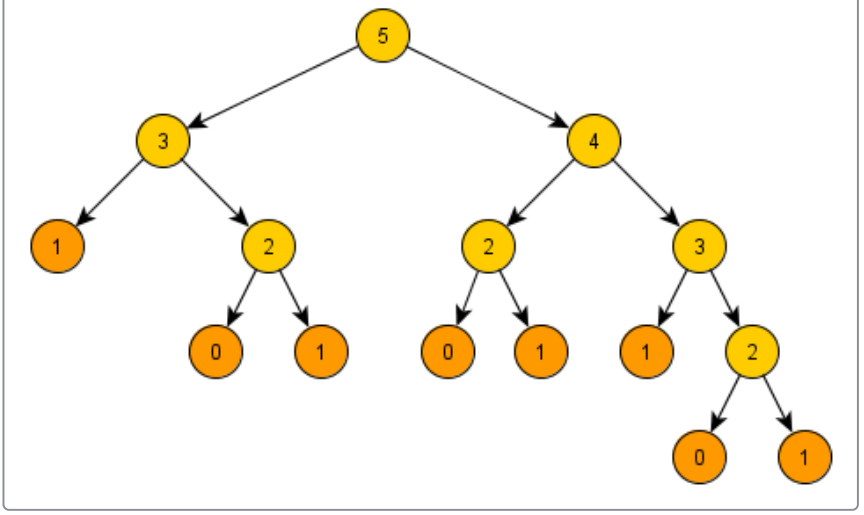
```java
/**
 * Fibonacciho posloupnost rekurzivne
 * @param index poradi cisla (pocitano od 0)
 * @return Fibonacciho cislo na danem poradi
 */
public static int fibonacciRek(int index){
    if(index == 0) return 0; //zarazka
    else if(index == 1) return 1; //druha zarazka
    else return fibonacciRek(index - 1) + fibonacciRek(index - 2);
       //rekurzivni volani
}
```

Co je špatně na rekurzivní implementaci

Fibonacciho posloupnost je oblíbeným chytákem při náborových pohovorech na pozici programátora. Nejde ani o to, že by to takto nešlo naprogramovat, kód je samozřejmě správně, ale o složitost dané implementace.

Zásadním problémem je právě rekurze. Pokud počítáme rekurzivně posloupnost v hodnotě 5, tak hodnotu 3 spočítáme 2x, hodnotu 2 spočítáme 3x a na triviální hodnoty se podíváme dokonce 8x. Už nyní je vidět obrovská nehospodárnost, která se s vyšší hodnotou parametru volání funkce bude dále jen zhoršovat.

Z implementačního hlediska je správně pouze iterativní verze tohoto algoritmu (*osmý díl* tohoto seriálu).



Fibonacciho posloupnost

Debugger

V tutoriálech Javy jsou často opomenuty nástroje, které výrazně usnadňují programování. Jedním z těchto nástrojů je i *debugger*, který slouží především k odhalování chyb. Debugger je integrován prakticky ve všech vývojových prostředích, nejinak je tomu i v *Netbeans IDE*.

Jak se používá a k čemu slouží?

Debugger nám umožňuje zastavit program na libovolném příkazu (program se zastaví před provedením tohoto příkazu) pomocí tzv. *breakpointu*. V Netbeans breakpoint umístíme kliknutím na číslo řádky ve sloupci nalevo od editoru. Po kliku by se měl zobrazit červený čtvereček, který znázorňuje jeho přítomnost. Pokud nyní program pustíme v *debug* módu (zvolíme v kontextovém menu projektu *Debug*), tak se program zastaví právě na zvolené řádce.

Nyní můžeme jednak program krokovat – postupovat po jednotlivých příkazech a zkoumat, kudy vlákno zpracování prochází, druhak se můžeme v každém kroku podívat na hodnoty jednotlivých proměnných a výrazů (záložka *Variables*).

Poslední významnou vlastností debuggeru je možnost změnit kód za běhu programu. K této schopnosti se ovšem vážou velká omezení. Nelze přidávat metody ani měnit jejich signatury a změna se projeví až při příštím průchodu metodou (tj. nelze měnit metodu s okamžitou platností uprostřed jejího vykonávání). Pokud chceme promítnout změny v kódu, tak musíme použít volbu *Apply Code Changes* v menu *Debug*.

Zkuste si sami projít vykonávání rekurzivní implementace faktoriálu a Fibonacciho posloupnosti s ohledem na postupný výpočet hodnoty.

– Pavel Mička
Zakladatel a administrátor stránek Algoritmy.net