Jazyk Java

Jak jsem potkal Javu (2)

V minulém čísle jsme si začali povídat o překvapeních, která čekají céčkaře, když se pustí do programování v jazyce Java, a o úvahách, které ho přitom mohou napadnout. Dnes se k tomuto tématu ještě jednou vracíme.

Objekty

Java zná pouze dynamické instance objektových typů. Zápis

Alfa a, b;

který v C++ znamená deklaraci proměnné třídy Alfa, a tedy mimo jiné volání konstruktoru, představuje v Javě pouze deklaraci odkazu (reference, tedy vlastně ukazatele) na instanci. Tento odkaz je třeba před použitím inicializovat, přidělit mu hodnotu, např. příkazem

a = new Alfa();

Zde pomocí příkazu new zavoláme konstruktor třídy Alfa bez parametrů, který vytvoří novou instanci, a její adresu uložíme do proměnné a. Odtud se odvíjí i řada dalších, pro céčkaře nezvyklých konstrukcí a problémů. Např. přiřazení

b = a;

nevytvoří kopii objektu, ale kopii odkazu – jak a, tak b budou odkazovat na týž objekt. Pokud chceme vytvořit skutečnou kopii objektu, musíme použít metodu clone, kterou všechny objekty dědí od společného předka, třídy Object.

Pokud bychom si v Javě chtěli vytvořit jednosměrný seznam celých čísel, bude deklarace třídy představující jeho prvek vypadat nejspíš nějak takto:

```
class Prvek {
   int data;                    // Data uložená v prvku
   Prvek dalsi;      // Následující prvek
   public Prvek(int x){data = x; dalsi = null;}
}
```

Na první pohled to vypadá velmi podivně – instance třídy prvek obsahuje sama sebe (následující prvek v seznamu), což se zdá technicky nemožné. Ve skutečnosti jde ovšem pouze o odkaz na tento prvek (tedy vlastně ukazatel), a to je naprosto v pořádku.

Správa paměti

Dalším závažným rozdílem mezi Javou a C++ je, že jednou vytvořený objekt nemůže programátor sám zničit (přesněji uvolnit z paměti). Už víme, že instance musíme alokovat dynamicky, pomocí new. Jednou vytvořený objekt existuje, dokud na něj ukazuje alespoň jeden odkaz. Po zániku posledního odkazu jej odstraní automatická správa paměti, tzv. garbage collector (což je docela výstižné pojmenování – doslova to znamená sběrač smetí nebo, chcete-li, popelář). To je na jedné straně obrovské dobrodiní, neboť v Javě prostě neexistují problémy s nevrácenou pamětí. (Pokud v C++ zapomenete vrátit alokovanou paměť, je do konce běhu programu ztracena; v Javě nic podobného nehrozí.)

Tím se také mění i zacházení s dynamickými datovými strukturami. Jestliže si např. vytvoříme jednosměrný seznam z instancí třídy Prvek, tj. datovou strukturu, která bude začínat odkazem prvni na první prvek, první prvek bude obsahovat odkaz na druhý prvek, druhý na třetí atd., pak ke smazání celého seznamu postačí jediný příkaz

první = null;

Hodnota null představuje “referenci nikam”, takže tím zrušíme odkaz na první prvek seznamu. Ten proto zanikne (předpokládáme, že jiný odkaz na tento prvek neexistuje). Jenže zánikem prvního prvku zanikne jediný odkaz na druhý prvek seznamu, takže zanikne i on – atd. Je to velice elegantní.

Na druhé straně ovšem algoritmy pro automatickou správu paměti nebývají právě efektivní, a pokud se na ně spolehneme, dočkáme se občas podivného chování: aplikace čas od času “nevysvětlitelně” zpomalí a po chvíli se zase vrátí k původnímu tempu. Jsou aplikace, kde to opravdu vadí. (Ostatně, pokud se pamatuji, garbage collector a jeho mizerná efektivita v jazyce Simula byly

jednou z příčin, proč se B. Stroustrup rozhodl vytvořit opravdu efektivní objektový jazyk – C++.)

Instance si tedy vytváříme sami, ale ruší je systém. Proto mají třídy konstruktory, ale nemají destruktory. Destruktor ve stejném smyslu jako v C++ by zde neměl velký smysl, neboť přesný okamžik zániku instance nelze předem určit. Pokud ovšem potřebujeme s instancí před zánikem něco provést, můžeme přetížit zděděnou metodu finalize. JVM ji zavolá předtím, než instanci zruší.

Operátor ->

Už jsme si řekli, že proměnné objektových typů v Javě představují odkazy, tedy ukazatele. Ovšem ke složkám objektových typů přistupujeme vždy pomocí operátoru . (tečka). Pokud delší dobu používáte Javu a pak se vrátíte k C++, budete možná – podobně jako já – zpočátku všude psát tečku.

To mne ale přivedlo na myšlenku, že operátor -> je v jazyce C vlastně zbytečný, šlo by ho nejspíš bez problémů nahradit tečkou. K záměně by nemohlo dojít, neboť význam by byl určen typem levého operandu (objekt nebo ukazatel na objekt). Přetěžování – tedy více významů jednoho operátoru – není na této úrovni v jazyce C nic neobvyklého, vzpomeňme si jen na operátor *, který podle okolností znamená násobení nebo dereferencování, nebo na &, který znamená bitové AND nebo získání adresy.

V C++ bychom ovšem odstraněním šipky přišli o jeden z programátorsky přetěžovatelných operátorů.

Je přetěžování operátorů nebezpečné?

Jednou z vymožeností, které bude programátor zvyklý na C++ postrádat, je přetěžování operátorů. Vzhledem k tomu, že jde o záležitost syntakticky naprosto neproblematickou, nezbývá než se dohadovat, že ji autoři Javy pokládali za nebezpečnou; jinak si neumím představit, proč ji do jazyka nezahrnuli. Je to jasný krok zpět. Nesmím-li přetěžovat operátory, musím je nahradit funkcemi a psát např.

Plus(Krat(a, b),c)

místo

a*b+c

O tom, co je přehlednější, asi není třeba mluvit. Mimochodem, na představu, že přetěžování operátorů je potenciálně nebezpečné, jsem narazil už několikrát. (Dokonce prý platí nařízení, že v programech používaných k obsluze jaderných zařízení se přetěžování operátorů nesmí používat. Setkal jsem se s tím při obhajobě jedné diplomové práce, ovšem příslušný předpis jsem neviděl.)

Občas se v této souvislosti setkávám i s úvahami o efektivitě, ale ty zde nejsou na místě: za prvé, použití přetíženého operátoru je ekvivalentní volání funkce (takže složitější je nejvýše překlad), a za druhé, kdyby šlo autorům Javy alespoň ve druhé nebo ve třetí řadě o efektivitu, asi by nevytvořili interpretovaný jazyk.

Zde Java navíc není důsledná: nejenže dovoluje přetěžování funkcí, ale navíc je ve standardní třídě String pro práci se znakovými řetězci přetížen operátor + pro zřetězení. Takže tvůrce jazyka smí něco, co programátor ne – a to není pěkné, i když mimo svět C a C++ je to poměrně obvyklé.

Zajímavé je i omezení operátoru čárka. V Javě ho totiž můžeme používat pouze v prvním a třetím výrazu v příkazu for.

(Ovšem možná že se přetěžování operátorů v Javě přece jen někdy dočkáme; mezi rezervovanými slovy totiž najdeme i operator.)

Funkce a jejich parametry

Jazyk Java umí předávat parametry pouze jedním způsobem, a to hodnotou. Ovšem předáme-li hodnotou odkaz na objekt, předáme vlastně objekt odkazem, takže ve skutečnosti se parametry primitivních typů předávají hodnotou a parametry ostatních typů (objektů a polí) odkazem. Pokud bychom chtěli, aby funkce (metoda) měnila hodnotu parametru primitivního typu, musíme tento parametr “zabalit” do některé z pomocných tříd (Integer, Double atd.)

To koneckonců ještě není tak zlé; legrace nastane ve chvíli, kdy chceme předat jako parametr funkci. To je potřeba zejména ve vizuálních vývojářských nástrojích při definici handlerů událostí.

Java nezná ukazatele, ani ukazatele na funkce, a nezná také funkcionální typy, jak je zavedl Pascal. (Ostatně funkcionální typy v Pascalu jsou jen jinak pojmenované ukazatele.) Chceme-li předat funkci jako parametr, musíme ji “zabalit” do vhodné třídy.

Podívejme se, jak se ke třídě jButton přidává nový “posluchač”, tj. metoda, která se bude volat při stisknutí tlačítka.

```
jToggleButton1.addActionListener(
new java.awt.event.ActionListener()
 {
    public void actionPerformed(ActionEvent e) {
     jToggleButton1_actionPerformed(e);
   }
});
```

O přidání posluchače se stará metoda addActionListener, které jako parametr předáme novou instanci třídy ActionListener; tuto instanci inicializujeme pomocí bezejmenné instance bezejmenné třídy, obsahující jedinou metodu, a to “handler”, který se má v případě potřeby volat. (Poznamenejme, že jde o kód automaticky generovaný JBuilderem. Podobné triky ovšem používají i jiná vývojová prostředí.)

Výjimky

Mechanismus výjimek vypadá velmi podobně jako v C++. Skutečnost, že všechny objekty, které slouží k přenosu informací o výjimkách, musí být potomky téže třídy (Throwable), není příliš překvapující; koneckonců standard C++ dospěl k něčemu podobnému, zůstalo však pouze u doporučení. Zpočátku může trochu problémy působit skutečnost, že pokud se z nějaké metody může rozšířit výjimka, musíme to explicitně deklarovat pomocí fráze throws v hlavičce – ovšem jen u výjimek odvozených od třídy Exception, nikoli však od tříd odvozených od jejího potomka RuntimeException. To mi připadá jako krásná ukázka situace, kdy čistota jazyka musí ustoupit praktickým ohledům (výjimky těchto typů jsou příliš časté, a tak se jaksi “mlčky předpokládají”).

Docela příjemná je i možnost připojit k bloku try blok finally – koncovku, která se provede vždy, ať blok try skončí jakkoli. (Podobnou konstrukci najdeme v jazyce C pro Win32 u tzv. strukturovaných výjimek.) V Javě – podobně jako v C – je to ale v podstatě železná nutnost: objekty v Javě zanikají až ve chvíli, kdy si systém usmyslí zavolat garbage collector, takže volání metody finalize nemůže nahradit automatické volání destruktoru.

V C++ se můžeme v převážné většině situací spolehnout na to, že víme, kdy instance zanikne, a proto si můžeme být jisti, kdy se zavolá destruktor. Můžeme mu tedy svěřit “úklidové” úkoly. Tato možnost v Javě chybí, a proto nezbývalo než nabídnout programátorům koncovky bloků, tj. úseky kódu, které se provedou vždy, i v případě, že v bloku try vznikne výjimka.

Vlákna

Ať už hovoříme o multithreadingu nebo o vícevláknovém zpracování, vždy to znamená totéž – možnost rozdělit výpočet do několika paralelních větví, které běží zároveň. Zde Java skutečně ve srovnání s C++ získává body, neboť multithreading řeší na úrovni jazyka, a tedy způsobem, který má naději být přenositelný. Základem je třída Thread a rozhraní Runnable. V obou případech musíme implementovat metodu run(), která bude představovat “tělo” vlákna.

Při vícevláknových výpočtech je vždy hlavním problémem synchronizace přístupu ke sdíleným datům nebo prostředkům. Java to umožňuje řešit pomocí tzv. synchronizovaných metod (deklarují se s pomocí klíčového slova synchronized). Dvě synchronizované metody téhož objektu nemohou běžet zároveň ve dvou různých vláknech.

Java zde vlastně využívá monitor, nástroj, který zabezpečuje, že jím hlídaný úsek kódu provádí vždy pouze jedno vlákno. Ovšem tento monitor není programátorovi přímo dostupný, může ho využívat pouze prostřednictvím synchronizovaných metod.

Jednotlivá vlákna mohou mít různé priority, lze je pozastavit, uspat, probudit atd. Pro ovládání vláken je k dispozici řada metod. Celé to ovšem má jednu vadu na kráse: některé z metod pro ovládání vláken jsou označeny jako nevhodné (deprecated). Bohužel se jedná zejména o metody, po kterých programátor, zvyklý na prostředí Win32, sáhne jako první (suspend aj.).

Ještě jednou čeština

V minulém dílu tohoto povídání jsem si stěžoval na problémy s češtinou Javě 2. Od té doby přece jen došlo ke změně; v nejnovější verzi, JDK 1.2.2, se kterou se setkáme např. v JBuilderu 3.5, sice

stále chybí soubory font.properties.cz nebo font.properties.sk, ale pod Windows NT 4.0 nebo Windows 2000 si aplikace umějí převzít nastavení z operačního systému. Pod Windows 98 je k dispozici alespoň částečné řešení založené na souboru font.properties.cz, ovšem, pokud vím, umí jen písmo Tahoma. Windows 95 jsou z tohoto hlediska stále jaksi mimo hru…

Takže “jiný kafe”...

Na první pohled by se leckomu mohlo zdát, že přechod od C++ k Javě je naprosto bez problémů. Jak jsme si ukázali, není to ale tak docela pravda, neboť v mnoha případech je sice syntaxe Javy (způsob zápisu programových konstrukcí) velice podobná jazyku C++, ale sémantika (jejich význam) se liší. Nezbývá tedy než se Javu opravdu naučit.

Miroslav Virius