Java a C++

Co má Java proti C++

V minulých dvou dílech článku "Jak jsem potkal Javu" jste si mohli přečíst o prvních dojmech céčkaře, který se seznamuje s jazykem Java. Zde na ně volně navážeme a podíváme se na některé chyby, které může vzájemná podobnost těchto jazyků způsobit. Můžeme se s nimi setkat při převodu programu z Javy do C++, nebo když se při programování v C++ prostě necháme Javou příliš ovlivnit.

Podobně jako předchozí články, i tento vychází z pohledu céčkaře. Příklady, které si zde ukážeme, mohou vypadat uměle, vznikly ale zkrácením skutečných programů.

C++ a Java

I když se občas setkáme s tvrzením, že Java vznikla z C++ vypuštěním neobjektových a potenciálně nebezpečných vlastností, není to úplná pravda. Syntaxe Javy byla jazykem C++ opravdu inspirována, nicméně sémantika (význam) některých konstrukcí se může v obou jazycích poněkud lišit, a proto také způsob programátorského uvažování se v obou jazycích liší dosti podstatně. Vztah obou jazyků znázorňuje připojený obrázek; vidíme na něm, že existují konstrukce, které jsou shodné (nebo velmi podobné) v obou jazycích, ale mají v každém z nich odlišný význam – a ty mohou být zdrojem problémů.

Polymorfismus

V Javě je pozdní vazba automatická. To znamená, že jestliže v potomkovi definujeme metodu, která bude mít stejné jméno, stejný počet parametrů stejných typů ve stejném pořadí a stejnou návratovou hodnotu jako metoda v předkovi, bude s ní program zacházet, řečeno terminologií C++, jako s metodou virtuální. Při jejím volání se bude uplatňovat pozdní vazba: i když k volání použijeme referenci na předka, zavolá program metodu odpovídající skutečnému typu instance.

V C++ si ovšem musíme použití pozdní vazby explicitně vyžádat pomocí klíčového slova virtual, které musíme uvést alespoň v bázové třídě. Na to lze při převodu programu z Javy do C++ snadno zapomenut a vzniknou docela zajímavé a nepříjemné chyby.

Přetěžování a dědění

Přetěžování se v Javě týká pouze metod, neboť tento jazyk neobsahuje globální funkce. Nicméně i zde můžeme narazit na situaci, kdy téměř stejná konstrukce vede v Javě k jiným výsledkům než v C++. Podívejme se na příklad, ve kterém definujeme třídu A a v ní metodu f(double). Od třídy A odvodíme potomka, třídu B, a v ní definujeme metodu f(int). (Má jiný typ parametru, nejde tedy o překrytí pro účely pozdní vazby, ale o přetížení.)

```
class A { // Java
   public void f(double d)
   {/*...*/}
}

class B extends A {
   public void f(int i)
   {/*...*/}
}

public class Pokus {
 public static void
 main(String[] argv) {
   B b = new B();
   b.f(3.5); // Zavolá se A.f(3.5)
   b.f(3);     // Zavolá se B.f(3)
 }
}
```

V metodě main() vytvoříme instanci b třídy B a pro ni zavoláme metodu f() jednou s

parametrem typu double, podruhé s parametrem typu int. Výsledkem je, že se v prvním případě zavolá zděděná metoda, ve druhém případě metoda definovaná ve třídě B. Zděděné metody jsou tedy v Javě na stejné úrovni jako metody definované v odvozené třídě.

Podívejme se nyní na analogický program v C++:

```
class A {   // C++
  public: void f(double){/*...*/}
};

class B: public A {
  public: void f(int){/*...*/}
};

int main()
{
  B* b = new B;
  b -> f(3.5); // volá se B::f(3)
  b -> f(3);     // opět B::f(3)
  return 0;
}
```

I když vše vypadá téměř stejně, výsledek se bude lišit: oba komentářem označené příkazy ve funkci main() způsobí volání B::f(). Důvodem je, že v C++ představuje každá třída samostatný obor viditelnosti a potomek je obor viditelnosti vnořený do předka. To znamená, že zděděná metoda A::f(double) je ve třídě B zastíněna metodou B::f(int). Při volání b -> f(3.5) se tedy překladač vůbec o volání zděděné metody nepokusí, uvažuje jen o metodách definovaných v potomkovi.

Pokud bychom v C++ trvali na volání zděděné metody, musíme si vypomoci plnou kvalifikací, např.

```
b -> A::f(3.5);
```

Poznamenejme, že mnohé z problémů, které z rozdílného zacházení s přetížením v odvozené třídě mohou vzniknout, zachytí překladač. Zde ovšem bylo možno konvertovat skutečný parametr udaný při volání na typ formálního parametru metody definované v potomkovi, a proto šlo uvedený program přeložit.

Ještě jednou přetěžování

Co když se typ skutečného parametru při volání přetížené metody neshoduje s typem parametru žádné z deklarovaných metod? Pak se překladač pokusí najít takovou metodu, pro kterou bude převod skutečného parametru na typ formálního parametru v nějakém smyslu co nejsnazší. Ovšem i zde se oba jazyky liší. Podívejme se opět na příklad:

```
public class Pokus {   // Java
 static void f(long s)
   {/* ... */}
 static void f(float s)
   {/* ... */}

 public static void
 main (String[] arg) {
   byte b = 3;
   f(b);   // Volá se f(long)
 }
}
```

Tento příklad se v Javě bez problémů přeloží a program zavolá metodu f() s parametrem typu long(). V C++ je situace jiná – následující konstrukci překladač označí za nejednoznačnou:

```
void f(long); // C++
void f(float);
//...
char c = 3;
f(c);   // nejednoznačné
```

V Javě je to poměrně jednoduché: číselné typy jsou seřazeny do posloupnosti byte, short, int, long, float, double. Čím více doprava, tím je rozsah typu větší. Převody na typ s větším rozsahem (tzv. rozšiřující konverze) mohou v Javě proběhnout automaticky. Pokud neexistuje metoda, která by měla formální parametr stejného typu, jako je skutečný parametr při volání, vybere se taková, na

jejíž typ lze skutečný parametr převést nejsnazší rozšiřující konverzí.

V C++ je daleko více možných konverzí – je zde mnohem více číselných typů, automaticky mohou proběhnout i zužující konverze, jsou možné i uživatelem definované konverze. Všechny možné konverze jsou rozděleny do několika skupin a konverze v každé z těchto skupin se pokládají za stejně obtížné. Například konverze typu char na long a na float se pokládají za stejně obtížné, a proto označí překladač C++ uvedenou ukázku za nejednoznačnou. Naproti tomu následující příklad je pro překladač C++ jednoznačný:

```
void f(long); // C++
void f(int);
//...
char c = 3;
f(c);   // zavolá se f(int)
```

Zde se zavolá f(int), neboť konverze typu char na int spadá do tzv. celočíselných rozšíření a pokládá se za jednodušší než konverze char na long.

Poznamenejme, že zužující konverze, tj. převod na typ s menším rozsahem, nejsou v Javě automatické. To znamená, že i zcela jednoznačné příklady z C++ může překladač Javy označit za chybné a požadovat v nich explicitní uvedení konverze (přetypování) parametru.

Přetypování a rozhraní

Jestliže v Javě třída X implementuje rozhraní Y a je potomkem třídy Z, lze s ní pracovat pomocí reference na Y i pomocí reference na Z. Přitom referenci na Z lze přetypovat na referenci na Y a naopak, například kvůli volání metody:

```
class X extends Z implements Y
{/* ... */}

Z z = new X();
(Y)z.f();     // OK
```

Analogií javských rozhraní v C++ jsou abstraktní třídy, které mají pouze virtuální metody. Ovšem pozor, přetypování pomocí operátoru (typ) zde nebude fungovat, program zavolá nesprávnou metodu nebo se zhroutí.

```
class X: public Z, public Y
{/* ... */}

Z *z = new X;
(Y*)z.f();     // Chyba
```

Má-li takovéto volání fungovat korektně, musíme použít operátor dynamic_cast. (Podrobnější rozbor najdete v pojednání “Přátelské nedorozumění nad kávou” – viz poznámku v rámečku.)

Specifikace výjimek

Jestliže v Javě v hlavičce metody neuvedeme žádnou specifikaci výjimek, znamená to, že se z této metody nesmějí rozšířit žádné výjimky (kromě typů odvozených od třídy RuntimeException). Neuvedeme-li žádnou specifikaci výjimek v C++, znamená to, že se z dané funkce může rozšířit jakákoli výjimka. Problémy, které z toho mohou vzniknout, se bohužel v C++ projeví až za běhu, zatímco v Javě je většinou odhalí překladač.

Volání konstruktoru

Třídy mohou mít v obou jazycích několik konstruktorů. Pokud se ve všech opakuje určitá činnost, např. inicializace skupiny proměnných, je v Javě poměrně běžné svěřit ji jednomu z konstruktorů a ten pak volat z ostatních. Jako příklad si definujme třídu Cplx, která bude reprezentovat komplexní čísla.

```
public class Cplx { // Java
   double re, im;
   public Cplx(double _re, double _im)
    {re = _re; im = _im;}
   public Cplx() {this(0,0);}
   public Cplx(double r){this(r, 0);}
}
```

První konstruktor má dva parametry a vytvoří komplexní číslo z dvojice reálných čísel. Druhý je

konstruktor bez parametrů, který vytvoří komplexní nulu – k uložení nul do složek použije konstruktor se dvěma parametry, kterému předá nuly; volá ho pomocí klíčového slova this. Třetí konstruktor vytvoří komplexní číslo z jednoho reálného (jako imaginární část doplní nulu); přitom také využije služeb konstruktoru se dvěma parametry. Takovéto použití konstruktoru je v Javě naprosto v pořádku a není nijak neobvyklé.

Nyní se podívejme na analogickou konstrukci v C++. Zde musíme konstruktor volat jménem třídy:

```
class Cplx {   // C++
  double re, im;
public:
  Cplx(double _re, double _im)
  :re(_re), im(_im){}
  Cplx(double _re){Cplx(_re, 0);}
};
```

Jeden z konstruktorů jsme vynechali, ale to na věci nic nemění. Podívejme se, co se stane, napíšeme-li

```
Cplx C(9);   // (1)
```

Bude-li C globální instance, budou obě její složky obsahovat 0, bude-li to dynamická nebo lokální automatická instance, budou její složky obsahovat nějaké podivné hodnoty. Volání jednoho konstruktoru v těle jiného konstruktoru téže třídy nezpůsobí provedení těla zavolaného konstruktoru pro danou instanci, ale vytvoření nepojmenované dočasné instance, která vzápětí zanikne.

Podívejme se na celou věc podrobněji. Příkaz (1) způsobí vyhrazení paměti pro instanci C a zavolání konstruktoru třídy Cplx s daným parametrem. V těle tohoto konstruktoru se provede příkaz Cplx(9,0). To znamená, že se zde vytvoří lokální nepojmenovaná instance třídy Cplx obsahující hodnoty 9 a 0. Tato instance ihned zase zanikne – a to je vše, paměť vyhrazená pro instanci C zůstane nedotčena.

Poznamenejme, že v této situaci by v C++ bylo daleko rozumnější využít implicitních hodnot parametrů. Konstruktor

```
Cplx::Cplx(double _re=0, double _im=0)
  :re(_re), im(_im){}
```

lze volat bez parametrů, s jedním nebo se dvěma parametry a vždy se zachová tak, jak potřebujeme.

Pokud bychom chtěli za každou cenu využít v jednom konstruktoru služeb jiného konstruktoru, museli bychom si vytvořit pomocnou instanci a tu pak přiřadit aktuální instanci, např. takto:

```
Cplx(double _re)
{*this=Cplx(_re, 0);}
```

To je ale trochu krkolomné řešení; lepší je naprogramovat část společnou pro všechny konstruktory v samostatné soukromé metodě a tu volat ze všech konstruktorů. Tento způsob lze samozřejmě použít i v Javě.

Inicializace datových složek

Datové složky objektů můžeme v Javě inicializovat přímo v deklaraci třídy nebo v konstruktorech. Složky, které explicitně neinicializujeme, budou mít hodnotu 0. (To znamená, že volání konstruktoru this(0,0) v ukázce v předchozím odstavci je vlastně zbytečné.) Řada programátorů s tím počítá a inicializace hodnotou 0 vynechává.

Pokud ovšem vynecháme inicializaci některé nestatické datové složky v C++, bude situace složitější. Neinicializované datové složky globálních instancí a lokálních statických instancí budou také mít hodnotu 0. Ovšem hodnoty neinicializovaných datových složek automatických a dynamických instancí nebudou definovány, což znamená, že v nich může být cokoli. (Z hlediska přechodu od Javy k C++ jsou důležité především dynamické instance, neboť Java ani jiné nezná.) Důsledky takového opomenutí v C++ si lze snáze představit než popsat.

Celá čísla

Java má čtyři celočíselné typy, všechny se znaménkem, a jejich rozsah přesně vymezuje. Například typ long nabývá v Javě hodnot v rozmezí od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807 (až do trilionů, tedy 10^{18}), předpokládá tedy 64bitovou reprezentaci čísel tohoto typu.

Na druhé straně C++ rozmezí hodnot typu long nespecifikuje; v typické dnešní implementaci jazyka C++ je tento typ dvaatřicetibitový, tj. nabývá hodnot v rozmezí od -2 147 483 648 do

2 147 483 647 (to je zároveň nejmenší interval, který musí tento typ pokrývat).

Na podobné rozdíly můžeme narazit i v případě typu int¸ jehož rozsah je v Javě od -2 147 483 648 do 2 147 483 647. V C++ může být i menší, musí však být alespoň od -32 768 do 32 767 (to je typická hodnota v 16bitových prostředích.)

Pochopitelně to znamená, že některé výpočty, které proběhly bez problémů v Javě, mohou při mechanickém přenosu do C++ způsobit celočíselné přetečení a poskytnout chybné výsledky. (Poznamenejme však, že novější překladače C++ nabízejí často jako rozšíření také 64bitová celá čísla, zpravidla pod označením long long nebo __int64. S jejich pomocí lze případné problémy snadno obejít.)

Reálná čísla

Java obsahuje dva datové typy pro reprezentaci reálných čísel, které se jmenují, stejně jako v C++, float a double. Protože jsou založeny na dnes běžně používané reprezentaci reálných čísel v procesoru, jsou – alespoň na první pohled – shodné s odpovídajícími typy v C++. Problémy může způsobit skutečnost, že ve výrazech těchto typů se mohou v Javě objevit jako operandy i jako výsledky speciální hodnoty nekonečno a NaN, zatímco v C++ v některých implementacích mohou, ale v jiných nemusejí.

To znamená, že v Javě můžeme napsat

```
float f = 1, g = 0;
float h = f/g; // nekonečno
float x = h*g; // NaN
```

a výsledkem dělení nulou bude plus nekonečno, zatímco ve většině implementací C++ způsobí takovéto dělení běhovou chybu.

Lze samozřejmě namítnout, že výsledky operací, při nichž se dělí nulou (nebo obecně jakýchkoli operací, při kterých dojde k přetečení v pohyblivé řádové čárce), nemají s největší pravděpodobností žádný rozumný smysl a že je lepší, když program ohlásí chybu, než aby s nesmyslnými mezivýsledky počítal dál. To je jistě pravda, nicméně lze si představit i rozumná použití takovýchto hodnot, třeba jen k signalizaci výpočtové chyby bez ukončení programu.

Poznamenejme zde, že kladné a záporné nekonečno (infinity) a NaN jsou zvláštní hodnoty reálných čísel požadované standardem IEEE 754. Kladné nekonečno může vzniknout např. při dělení kladného reálného čísla nulou, při přetečení atd. a chová se podobně jako nekonečno v matematice. To znamená, že přičteme-li k nekonečnu jakékoli konečné číslo, dostaneme opět nekonečno, vydělíme-li jakékoli konečné číslo nekonečnem, dostaneme nulu ap.

NaN je zkratka slov Not a Number (“nečíslo”) a vznikne při operacích, které nemají smysl – např. při násobení nuly a nekonečna. Matematický koprocesor intelských procesorů s těmito hodnotami umí zacházet; jeho chování ovšem závisí na nastavení tzv. řídicího slova (control word), jehož jednotlivé bity mimo jiné určují, zda při přetečení vznikne výjimka procesoru, nebo zda bude výsledkem nekonečno.

Nepřekvapivý závěr

Každý programovací jazyk má svou vnitřní logiku; chceme-li ho správně používat, musíme v něm myslet (stejně je tomu ostatně i s jazyky národními...). Jinak brzy zjistíme, že děláme mnoho zbytečných chyb. To bohužel platí zejména v případě jazyků, které si jsou na pohled podobné, jako je Java a C++.

Miroslav Virius

Céčkaři a javaři

Série našich článků porovnávajících z pohledu céčkaře Javu a C++ vyvolala zajímavou odbornou polemiku; není asi náhodou, že tak trochu připomíná spory různých církví uctívajících (každá svým způsobem) stejného boha – v tomto případě objektové programování. Chcete-li se nad odlišnými přístupy k tomuto pojmu zamyslet hlouběji, na aktuálním Chip CD najdete v rubrice Chip Plus dva příspěvky, jejichž názvy vám nejspíš už předem prozradí “náboženské vyznání” autorů: “Silná káva pro objektového programátora” a “Přátelské nedorozumění nad kávou”.