PARAMETRIZOVANÉ DATOVÉ TYPY A METODY V JAZYKU JAVA 5.0

Rudolf Pecinovský

Amaio Technologies, Inc., Třebohostická 14, 100 00 Praha 10,
rudolf@pecinovsky.cz

Abstrakt
Jedním z nejvýznamnějších rozšíření poslední verze jazyka Java bylo zavedení parametrizovaných datových typů a metod. Příspěvek se zabývá některými jejich méně známými vlastnostmi, zejména pak vlastnostmi, jejichž důsledky mohou zavinit nečekanou reakci překladače či přeloženého programu. Naznačuje také možnosti použití metaznaků (žolíků), vlastnosti hierarchií parametrizovaných datových typů a zabývá se i spoluprací programů využívajících těchto typů s programy napsanými v předešlých verzích jazyka. V neposlední řadě pak rozebírá i dočasnost jejich platnosti a z toho plynoucí důsledky.

1 POUŽITÍ

Parametrizované typy a metody (dále PT, PM či souhrnně PTM), s nimiž přišla Java 5.0, jsou největší syntaktickou inovací od vzniku Javy. Hlavním účelem jejich zavedení byla snaha po zlepšení vyjadřovacích schopností a bezpečnosti pramenící z přesnější definice typů a možnosti přetypování.

PTM umožňují typovou kontrolu i v situacích, v nichž to doposud nebylo možné a programátoři většinou využívali implicitního přetypování na rodičovskou třídu či implementované rozhraní doprovázené pozdějším explicitní přetypováním zpět na vlastní třídu dané instance. PT umožňují přesunout řadu typových kontrol z doby běhu do doby překladu.

PTM jsou v podání Javy záležitostí překladače. Mohli bychom je označit jako typy či metody s typovými parametry, jimiž jsou třídy a rozhraní, které budou použity při práci s konkrétní instancí daného parametrizovaného typu, resp. metody.

PTM nevytvářejí rodiny typů a metod, jako je tomu u šablon C++. Jedná se pokaždé o týž základní typ, jehož typové parametry slouží pouze jako další informace pro překladač, jenž na jejich základě provádí některé dodatečné kontroly a/nebo automaticky vkládá potřebná přetypování.

Po vzoru jazyka C++ se typové parametry uvádějí ve špičatých závorkách. Při parametrizaci typu se uvádějí bezprostředně za názvem typu (a to i při volání příslušného konstruktoru), při parametrizaci metod jsou uváděny v hlavičce metody před uvedením typu její návratové hodnoty. Budeme-li chtít definovat proměnnou, která bude seznamem textových řetězců, definujeme ji příkazem:

```
List<String> seznam = new ArrayList<String>();
```

Jak si jistě všimnete, typový parametr je třeba uvést nejenom v deklaraci typu, ale také ve volání příslušného konstruktoru. Pokud je typových parametrů více, oddělují se čárkami:

```
Map<Integer,String> mis = new HashMap<Integer,String>();
```

2 DEFINICE VLASTNÍCH PTM

Při definici vlastních parametrizovaných typů uvádíme typové parametry ve špičatých závorkách za názvem typu. Přitom můžeme uvést i případná **omezení** (*boundary*), kterým musí typové parametry vyhovovat – konkrétně že musí být potomkem nějaké třídy nebo implementovat nějaká rozhraní.

Deklarovaný typový parametr pak můžeme v deklaraci vzápětí použít, např. při deklaraci předků či implementovaných rozhraní. Typ `Interval` můžeme definovat např. následovně:

```
public class Interval<T extends Comparable<T>> {
    T dolní, horní;
    public Interval( T d, T h ) { dolní = d; horní = h; }
    public T getDolní() { return dolní; }
    public T getHorní() { return horní; }
    public void setDolní( T dolní ) { this.dolní = dolní; }
    public void setHorní( T horní ) { this.horní = horní; }
    public boolean uvnitř( T t ) {
        return (dolní.compareTo(t) <= 0)  && (t.compareTo(horní) <= 0);
    }
    public String toString() { return "<" + dolní + ";" + horní + ">"; }

    public static <T extends Comparable<T>>
           void porovnej( Interval<T> in, T ... tt ) {
        System.out.println("\nInterval: " + in);
        for( T t : tt )
            System.out.println( t + " leží uvnitř: " + in.uvnitř(t));
    }
}
```

Jako poslední je v předchozí ukázce uvedena definice parametrické metody `porovnej`, která vypíše na standardní výstup informace o tom, které ze zadaných parametrů leží uvnitř intervalu zadaného jako první parametr.

Typové parametry mohou mít deklarováno několik omezení současně. Protože oddělovač čárka je použit pro oddělení jednotlivých typových parametrů, je pro oddělení jednotlivých omezení použit znak `&`. Mohli bychom např. definovat následující třídu:

```
public class VíceParametrůAOmezení
       <A, B extends Iterable<A> & Comparable<A>, C extends A> {
    A a;
    B b;
    C c;
}
```

Parametrizované typy mohou vytvářet obdobné dědické hierarchie jako běžné typy. Přitom PT může být jak rodičem, tak potomkem neparametrizovaného typu. V následující ukázce je parametrizovaný předek potomkem neparametrizované třídy `Object` a má dva neparametrizované potomky.

```
class Předek<T> extends Object { T pred; }

class PotomekI extends Předek<Integer> {
    PotomekI( Integer i ) { pred = i; }
}

class PotomekS extends Předek<String> {
    PotomekS( String s ) { pred = s; }
}
```

3 PARAMETRIZACE A OČIŠŤOVÁNÍ

Jedním z hlavních požadavků, které bylo při specifikaci rozšíření nové verze Javy nutno dodržet, bylo zachování zpětné kompatibility, které umožní vytvářet programy bezproblémově spolupracující s programy napsanými v dřívějších verzích.

Zachování této kompatibility bylo dosaženo především díky koncepci očišťování parametrizovaných datových typů od typových parametrů a jejich převodu do neparametrizovaného tvaru, s nímž si budou vědět rady i dříve napsané programy.

3.1 Terminologie

Než se o parametrizovaných datových typech rozhovořím podrobněji, zavedeme si nejprve několik pojmů:

Termínem **ozdobený datový typ** budu označovat datové typy doplněné o typové parametry. Za ozdobené bych tedy označil např. typy `List<String>` a `LinkedList<String>`.

Překladač zbavuje v průběhu překladu datové typy jejich případného zdobení. Tento proces budu označovat jako **čištění** nebo **očištění** (anglicky *erasure*) datových typů.

Očištěné nebo prostě jen neozdobené datové typy jsou ve specifikaci označovány jako **raw types**. Protože slovo *raw* lze překládat mnoha způsoby a v různých kontextech jsou různé z nich vhodné, budu datové typy bez přídavných typových parametrů označovat střídavě jako **surové**, **neozdobené** nebo **očištěné** (je to obráceně než u cukru – surový cukr je nečištěný). Za surové, neozdobené či očištěné datové typy bychom tedy označili např. typy `List` a `LinkedList`.

3.2 Očištění parametrizovaného kódu

Kdykoliv použijete v programu nějaký ozdobený datový typ, překladač jej v průběhu překladu od jeho ozdob očistí a do class-souboru jej uloží jako očištěný, surový typ. Místo neomezovaných typových parametrů pak dosadí typ `Object` a místo omezovaných dosadí první z uvedených omezujících typů. Třída

```
class PředČištěním <CS extends Comparable<CS> & Serializable,
                    SC extends Serializable & Comparable<SC>>  {
    CS cs;
    SC sc;
    void upravCS( CS p ) { if(cs.compareTo(p) < 0)  cs = p; }
    void upravSC( SC p ) { if(sc.compareTo(p) < 0)  sc = p; }
}
```

bude přeložena naprosto stejně, jako kdyby byla definována následovně:

```
class PoČištění {
    Comparable   cs;
    Serializable sc;

    void upravCS(Comparable p) {
        if(cs.compareTo((Object)p) < 0)  cs = p;
    }
    void upravSC(Serializable p) {
        if(((Comparable)sc).compareTo((Object)p) < 0)  sc = p;
    }
}
```

Z podoby poslední metody po očištění jasně vyplývá, že bychom čistě informační rozhraní (např. `Clonable`, `Serializable` apod.) měli deklarovat jako poslední, protože jinak by se nám program zbytečně hemžil různými přetypováními.

Při očišťování výrazů uvnitř programu záleží na tom, odpovídá-li deklarovaný typový parametr požadovanému typu.

Někdy ale překladač ohlásí chybu, která nás trochu zaskočí. Např. následující metodu odmítne přeložit, dokud z kódu neodstraníme zakomentovaný příkaz. Všimněte si přitom, že následující příkazy, pomocí nichž zjišťujeme tutéž informaci, ale v klidu přeloží.

```
public static <T extends Comparable<T>> void whoAmI( T a ) {
    System.out.println("Object: " + (a instanceof Object) );
    System.out.println("Comparable: " + (a instanceof Comparable) );
//    System.out.println("Double: " + (a instanceof Double) );
    System.out.println("Double 0: " + ((Object)a instanceof Double) );
    System.out.println("Double 1: " + (a.getClass() == Double.class) );
    System.out.println("Double 2: " + (Double.class.isInstance(a)) );
}
```

3.3 Přemosťovací metody

Odvodíte-li od PT nějakého potomka, může se stát, že pro něj budete potřebovat překrýt některou ze zděděných metod. Ne vždy ale budete mít k dispozici typový parametr rodičovské nebo dokonce prarodičovské třídy.

Definujme např. potomka třídy `Interval`, který bude pracovat pouze s textovými řetězci a bude je porovnávat bez ohledu na velikost znaků. Jeho definice by mohla vypadat následovně:

```
public class IntervalS extends Interval<String> {

    public IntervalS( String sd, String sh )  { super( sd, sh ); }

    public boolean uvnitř( String s ) {
        return (dolní.compareToIgnoreCase( s ) <= 0)  &&
               (horní.compareToIgnoreCase( s ) >= 0);
    }
}
```

Vše je v pořádku až do chvíle, kdy si uvědomíte, že třída `IntervalS` je vlastně potomkem třídy `Interval`, v níž je ale po očištění její jediná metoda definována jako `uvnitř(Comparable)`, kdežto v třídě `IntervalS` jsme ji definovali jako `uvnitř(String)`. Metoda proto původní metodu nepřekrývá, i když by ji podle ducha návrhu překrývat měla!

Řešením není definovat metodu jako `uvnitř(Comparable)`, protože pak bychom přišli o typovou kontrolu, kvůli které byly typové parametry a parametrizované typy do Javy zavedeny. Překladač proto postupuje obráceně. Definuje **přemosťovací metodu** (*bridge method*), která překryje původní metodu `uvnitř(Comparable)` a jejíž tělo bude tvořeno jediným příkazem: voláním nově definované metody `uvnitř(String)`.

Obdobný problém nastane, budeme-li definovat metodu, která bude vracet místo původního očištěného typu aktuální typ platný pro danou třídu. Tady je ale situace trochu složitější a není tu prostor ji rozebírat. Odkážu vás proto na **Chyba! Nenalezen zdroj odkazů.**.

4 ZAKÁZANÉ OPERACE

Koncepce PTM a čištění programu obsahujícího parametrizované typy a metody vede k zákazu některých operací:

4.1 Za typové parametry nelze dosazovat primitivní typy

Koncepce PTM vychází z toho, že skutečné typy parametrů mohou být spolehlivě převedeny na některý z omezujících typů, a není-li deklarován žádný omezující typ, tak na typ `Object`. Tuto možnost však primitivní typy neposkytují, a proto se v případě potřeby práce s hodnotami některého z primitivních typů musíme u typových parametrů spokojit s použitím příslušného obalového typu.

4.2 Typové parametry třídy není možno použít u statických členů

Deklarace typu statického atributu, typu návratové hodnoty či parametru statické metody vyvolá syntaktickou chybu. Statický atribut či metoda totiž nesmí záviset na žádné instanci svého parametrizovaného typu, a proto nesmí být v jeho definici použity typové parametry, protože ty může každá z instancí definovat jinak.

4.3 Nelze vytvořit instanci parametrizovaného typu

V přeloženém programu totiž nejsou dostatečné informace o typu, jehož instance se má vytvořit. Protože není předem známý typ instance, nelze předem zaručit, že třída bude nabízet požadovaný konstruktor. Instance můžeme získat dvěma způsoby:

- Metody, které s nimi mají pracovat, získají potřebné instance prostřednictvím svých parametrů.
- Potřebujeme-li získat danou instanci opravdu až v průběhu výpočtu, musíme použít reflexi.

4.4 Nelze vytvořit pole instancí parametrizovaného typu

Pole si pamatuje typ svých prvků, a to i poté, co pole přetypujeme. Pokusíme-li se do něj vložit hodnotu nekompatibilního typu, vyvolá výjimku `ArrayStoreException`. Toto chování však není možné po očištění od typových parametrů zaručit.

4.5 Omezení při používání výjimek

Parametrizované typy nemohou být potomky typu `Throwable`. Deklarace typu

```
public class MojeVýjimka<T> extends Exception
```

je považována za syntaktickou chybu. Program proto nemůže deklarovat vyhození výjimky parametrizovaného typu.

Při vyhození výjimky totiž není známo, kdo výjimku zachytí, a není mu proto možno jakýmkoliv způsobem předat informace, které jsou při očišťování od typových parametrů odstraněny. Parametrizace výjimky by proto neměla žádný smysl.

Potomek třídy `Throwable` sice nemůže být definován jako PT, ale může být deklarován jako typový parametr. Dokonce umožňuje vyhodit výjimku příslušného typu:

```
public void zkus( int param, T t )  throws T {
    if( (param & 1) == 0 )
        throw t;
}
```

5 NEJEDNOZNAČNOSTI A KOLIZE

Očistěním kódu se ztrácí spousta informací, takže se pak může stát, že zdrojový kód, který se zdá být na první pohled jednoznačný, se ve skutečnosti ukáže být plným nejrůznějších skrytých nejednoznačností a kolizí. Uvedu zde pouze stručný výčet některých možností. Podrobnosti najdete opět v **Chyba! Nenalezen zdroj odkazů.**.

5.1 Falešně přetížená metoda

Programátoři jsou zvyklí na to, že metody, které mají deklarované různé typy parametrů a které označujeme jako *přetížené*, definuje překladač jako nezávislé metody. Problém je ale v tom, že různé typové parametry ještě nemusejí znamenat různé datové typy. Definice třídy podobné té následující proto způsobí syntaktickou chybu:

```
public class FalešnéPřetížení<T1, T2> {
    //...
    public void proveď( T1 p )  { /* ... */ }
    public void proveď( T2 p )  { /* ... */ }
    //...
}
```

5.2 Nová metoda koliduje se zděděnou

Naprosto stejný případ nastane tehdy, pokusíme-li se metodu nevhodně přetížit, tj. pokusíme-li se definovat metodu se stejným názvem, jako má některá ze zděděných metod, avšak zdánlivě jiným typem parametru. Tento zdánlivě jiný typ se však po očištění ukáže jako ten, který je použit ve zděděné metodě. Překladač by ale obě metody potřeboval rozlišit. Protože je však rozlišit nemůže, ohlásí syntaktickou chybu.

5.3 Kolize požadovaných rozhraní

Implementuje-li rodičovská třída parametrizované rozhraní, musíme počítat s tím, že všichni její potomci budou toto rozhraní implementovat se shodnými typovými parametry. Bude-li tedy rodičovská metoda deklarována

```
class Rodič implements Comparable<Rodič>
```

bude její potomek implementovat také rozhraní `Comparable<Rodič>`. V řadě případů to nevadí, ale velmi často nastanou situace, kdy tato skutečnost vyvolá kolizi.

5.4 Kolize implementovaných rozhraní

Třída ani typový parametr nesmí vícenásobně implementovat stejné rozhraní, které by mělo v různých deklaracích různé typové parametry. Není proto možné deklarovat:

```
class Rodič implements Comparable<Rodič>

class Potomek extends Rodič implements Comparable<Potomek>
```

5.5 Špatné pochopení dědičnosti

Při práci s PT musíme dát pozor na to, abychom na ně nepřenesli naše zkušenosti z práce s poli a dynamickými kontejnery. Mezi instancemi tříd s typovými parametry totiž platí zcela jiná pravidla přípustnosti a nepřípustnosti vzájemného zastupování.

Vezměme si např. následující příklad, v němž vystupují dva seznamy, jejichž prvky váže vztah dědičnosti:

```
/* 1 */  List<String> str = new ArrayList<String>();
/* 2 */  List<Object> obj = str;  //Syntaktická chyba
/* 3 */  obj.add(new Object());   //...abychom nemohli udělat toto
/* 4 */  String s = str.get(0);
```

První řádek je zcela v pořádku. Na druhém řádku však překladač ohlásí syntaktickou chybu. Kdyby tak neučinil, tak bychom se po zdánlivě korektním přiřazení ve třetím řádku ukládali ve čtvrtém řádku do řetězcové proměnné odkaz na obecný objekt.

Obecně platí: *To, že je jeden typ potomkem nebo implementací druhého nijak neimplikuje obdobnou vazbu u parametrizovaných typů, v nichž dané typy vystupují jako typové parametry*.

6 ŽOLÍKY

Dva PT, které se liší pouze hodnotami svých typových parametrů, jsou považovány za vzájemně nezávislé bez ohledu na to, jsou-li jejich typové parametry nějak příbuzensky spřízněny. Tato skutečnost však při používání PT velmi svazuje ruce. Ve starších verzích Javy bychom definovali metodu pro tisk jednotlivých členů kolekce na samostatné řádky následovně:

```
public void tiskniKolekci( Collection c ) {
    for( Iterator it = c.iterator();   it.hasNext();   )
        System.out.println( it.next() );
}
```

Takto definovaná metoda nám umožní tisk libovolné kolekce. Definujeme-li však v „nové Javě“ zdánlivě ekvivalentní metodu:

```
public void tiskniKolekci( Collection<Object> c ) {
    for( Iterator it = c.iterator();   it.hasNext();   )
        System.out.println( it.next() );
}
```

můžeme s ní tisknout pouze kolekce obsahující prvky typu `Object`. Jak jsme si vysvětlili před chvílí, kolekci obsahující instance typu `Object` nemůžeme považovat za předka žádné jiné kolekce, takže je tato metoda pro jakoukoliv jinou kolekci nepoužitelná.

Aby nám PT nesvazovaly ruce svojí neschopností zavést vztahy ekvivalentní dědičnosti, zavádí Java 5.0 *zástupný typ* **?** (otazník), jenž označuje libovolný typ vyhovující případným omezujícím podmínkám. Tento typ pak funguje mezi typovými parametry obdobně jako **žolík** (*wildcard*), který hodnotu typového parametru v daném místě nijak neomezuje.

K parametrizovanému typu, jemuž jsme jako typový parametr dosadili žolík, se pak můžeme chovat jako k rodiči všech typů s konkrétními hodnotami typových parametrů, jenž za zástupný typ dosadí nějaký konkrétní typ. Naopak k PT s konkrétním typovým parametrem se

můžeme chovat jako k potomkovi odpovídajícího PT používajícím místo daného typu žolík. Kdybychom použili v našem předchozím příkladu žolík, získala by definice tvar:

```
public void tiskniKolekci( Collection<?> c ) {
    for( Iterator it = c.iterator();   it.hasNext();   )
        System.out.println( it.next() );
}
```

Takto definovanou metodu již můžeme použít k tisku libovolné kolekce bez ohledu na typ jejích prvků. Od metody používající surový, neozdobený typ `Collection` se bude lišit pouze tím, že nás překladač nebude zasypávat varovnými zprávami, upozorňujícími na potenciálně nebezpečné operace.

6.1 Omezení hodnot žolíků

I na žolíky můžeme klást dodatečná omezení, v nichž můžeme požadovat, aby byl typ zastupovaný žolíkem potomkem nebo (na rozdíl od typových parametrů) předkem definovaného typu.

Představte si, že bychom potřebovali definovat obecnou metodu, která by naplnila seznam náhodnými hodnotami zadaného typu, a další metodu, která by spočetla průměr těchto hodnot. Všechny třídy, s jejichž instancemi lze počítat, jsou potomky třídy `Number`. My však nemůžeme daný seznam deklarovat jako seznam instancí třídy `Number`, protože pak bychom do této proměnné nemohli ukládat odkazy na seznamy instancí jiných typů. Použijeme-li však žolíka, vše se rozběhne.

```
public void test()
{
    List<? extends Number> lin;
    lin = new ArrayList<Double>();   naplň( lin );   tiskni(průměr( lin ));
    lin = new LinkedList<Zlomek>();  naplň( lin );   tiskni(průměr( lin ));
}
```

V jiných situacích zase potřebujeme definovat žolíka, který by mohl zastupovat i kteréhokoliv z rodičů daného typu. V podkapitole 5.4 *Kolize implementovaných rozhraní* jsme si ukazovali, že implementuje-li rodičovská třída rozhraní `Rozhraní<Rodič>`, implementují toto rozhraní i její potomci, přestože by občas potřebovali implementovat spíše rozhraní `Rozhraní<Potomek>`. Pokud ale v metodě používající tohoto potomka místo parametru typu

```
<T extends Comparable<T>>
```

kterému instance potomků rodiče implementujícího rozhraní `Comparable<Rodič>` nevyhovují, deklarujeme parametr typu

```
<T extends Comparable<? super T>>
```

pak takto definovanému rozhraní již všichni potomci našeho rodiče vyhoví. Problematika je opět složitější a musím vás proto odkázat s podrobnostmi na uvedenou literaturu.

7 PARAMETRIZOVANÉ TYPY A REFLEXE

V nové verzi Javy je nyní parametrizovaná i třída `Class`. Např. objekt `String.class` je nyní instancí parametrizovaného typu `Class<String>`. Hlavní výhodou typového parametru je to, že typy parametrů a návratových hodnot metod instancí třídy `Class<T>` mohou nyní být mnohem přesněji specifikovány. Třída nyní definuje metody:

```
T newInstance()
T cast(Object)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class...)
```

a další. Např. o metodě `newInstance()` již nyní víme, že vrací přímo instanci požadovaného typu a nemusíme ji už proto přetypovávat.

Novinky jdou ale ještě dál. Nová verze přinesla celou plejádu nových typů, které slouží k detekci toho, že daný typ byl deklarován jako parametrizovaný. Při čištění se sice tyto informace z programu ztratí, ale v class objektech tříd a rozhraní zůstanou zachovány a lze je odtud vyloudit.

Tyto informace sice nepomohou zjistit, s jakými hodnotami typových parametrů byla vytvořena ta která instance, ale mohou prozradit, že třída či rozhraní byla definována jako parametrizovaná a mohou pomoci určit i podobu této definice.

LITERATURA

[1] PECINOVSKÝ Rudolf. *Java 5.0 – Novinky jazyka a upgrade aplikací*. Praha : Computer Press, 2005, ISBN 80-251-0615-2

[2] HORSTMANN Cay S.; CORNELL Gary. *Core Java, Volume 1 – Fundamentals.* Sun Microsystems Press, 2004. ISBN 0-13-148202-5.

[3] SHILDT Herbert. *Java: The Complete Reference, J2SE 5 Edition.* McGraw-Hill/Osborne, 2005. ISBN 0-07-223073-8.

[4] BRACHA Gilad. *Generics in the Java Programming Language.* Sun Microsystems, 2004, dostupný z WWW: <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>

[5] *JSR 14: Add Generic Types To The JavaTM Programming Language*. Sun Microsystems, 2004. dostupný z WWW: <http://jcp.org/en/jsr/detail?id=14>