Novinky, s nimiž přichází Java 5.0

Rudolf Pecinovský[1]

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

Abstrakt. Poslední verze jazyka Java přinesla nejpřevratnější změny v jeho historii. Tentokrát se nejenom rozšířila standardní knihovna, ale změny se výrazně dotkly i syntaxe jazyka, která doznala několika zásadních rozšíření. Příspěvek probírá syntaktické novinky, s nimiž přichází Java v edici J2SE 5.0. Jsou v něm postupně rozebrány statický import, automatické převádění primitivních hodnot na instance obalových typů a zpět, objektové výčtové typy, generické, resp. parametrizované datové typy, rozšíření příkazu for, zavedení metod s proměnným počtem parametrů a metadata.

Klíčová slova: Java 5.0, syntaxe

1 Základní cíle nového návrhu

Od vzniku jazyka (přesněji od jeho představení veřejnosti) v roce 1995 přinášely nové verze většinou pouze rozšíření standardní knihovny, i když někdy poměrně zásadní. Vlastní jazyk se však výrazných inovací nedočkal. Jediným významnějším rozšířením byla koncepce vnořených, vnitřních a lokálních tříd, s níž přišla v roce 1997 Java 1.1. Přidání klíčového slova **assert**, které v roce 2001 zavedla Java 1.4, bylo změnou spíše kosmetickou.

Zásadnější rozšíření jazyka přinesla teprve Java 5.0, která byla oficiálně uvedena letos 30. září. Všechna její rozšíření jsou motivována několika společnými cíli:

- Zvýšit přehlednost zdrojového kódu.
- Zvýšit spolehlivost, bezpečnost a výkon přeložených programů.
- Minimalizovat nekompatibilitu s předchozími verzemi.
  - Obejít se bez změn ve funkci virtuálního stroje.
  - Dosavadní programy musí bez problémů chodit.
  - Pokud možno nezavádět nová klíčová slova (bylo zavedeno pouze jediné nové klíčové slovo – **enum**).

Původním záměrem autorů bylo překládat programy do takového tvaru, aby je bylo možno spouštět i na starších verzích virtuálních strojů. Pracovaly tak dokonce i první beta verze. V závěrečných etapách vývoje produktu však bylo od tohoto záměru upuštěno.

2 Import statických atributů a metod

Pravděpodobně nejmírnějším a na druhou stranu take nejrozpačitěji přijímaným rozšířením je statický import. Ten umožňuje deklarovat statické atributy a metody jiných tříd, které bude možné v těle třídy používat bez nutnosti jejich kvalifikace.

Ve verzích do 1.4 včetně bylo nutno každé volání statického atributu nebo metody z jiné třídy kvalifikovat – např.:

```
import javax.swing.JOptionPane;

public class StatickýImport_Dříve
{
    private static int zlomek = 6;

    public static void dříve() {
        double sinus = Math.sin( Math.PI / zlomek );
        String odpoved = null;
        switch( JOptionPane.showConfirmDialog( null,
                    "Hodnota sin( PI/" + zlomek + " ) = " + sinus +
                    "\n\nJste se svym odhadem spokojen?" ) )
        {
            case JOptionPane.YES_OPTION:
                odpoved = "To mne tesi";    break;
            case JOptionPane.NO_OPTION:
                odpoved = "To mne mrzi";    break;
            case JOptionPane.CANCEL_OPTION:
            case JOptionPane.CLOSED_OPTION:
                odpoved = "Neutecete";      break;
        }
        JOptionPane.showMessageDialog( null, odpoved );
    }
}
```

Při použití Javy 5.0 stačí deklarovat import použitých statických členů, přičemž lze použít stejné hvězdičkové konvence jako u klasického příkazu **`import`**.

```
import static java.lang.Math.PI;
import static java.lang.Math.sin;
import static javax.swing.JOptionPane.*;

public class StatickýImport_Nyní
{
    private static int zlomek = 6;

    public static void nyní() {
        double sinus = sin( PI / zlomek );
        String odpověď = null;
        switch( showConfirmDialog( null,
                "Hodnota sin( PI/" + zlomek + " ) = " + sinus +
                "\n\nJste se svým odhadem spokojen?" ) )
        {
            case YES_OPTION:
                odpověď = "To mne těší";    break;
            case NO_OPTION:
                odpověď = "To mne mrzí";    break;
            case CANCEL_OPTION:
            case CLOSED_OPTION:
                odpověď = "Neutečete";      break;
        }
        showMessageDialog( null, odpověď );
    }
}
```

Hlavní výhradou oponentů proti statickému importu je ztráta informace o mateřské třídě použitého statického členu, která může ve čtenáři vyvolat dojem, že použité členy jsou statickými atributy či metodami dané třídy.

3 Automatické převody mezi primitivními a obalovými typy

Novinkou, která byla pravděpodobně inspirována obdobnou vlastností jazyka C#, je automatické převádění hodnot primitivních typů na instance příslušných obalových typů a zpět. (Tato vlastnost je v originální dokumentaci označována jako *autoboxing* a *auto-unboxing*). Část programu, kterou bychom museli ve starších verzích jazyka napsat následovně:

```
int     i3  = 3;
Integer tři = new Integer( i3 );
int     i6  = 2 * tři.intValue();
```

je nyní možno napsat jednoduše

```
int i3      = 3;
Integer tři = i3;
int i6      = 2 * tři;
```

Hlavní použití automatických převodů však nebude ve výrazech, ale při předávání hodnot parametrů metodám vyžadujícím parametry objektových typů, mezi nimi pak především metodám pro práci s dynamickými kontejnery.

Puristé mají výhrady i proti automatickým převodům, protože se díky nim ztrácí explicitní informace o tom, co je hodnotou primitivního typu a co je instancí objektového typu. Praktici zase namítají, že nyní ztrácejí přehled nad vytvářením nových instancí, což může vést k neočekávanému snížení efektivity programu.

4 Výčtové typy

Jednou z věcí, kterou kritici Javě vyčítali, byla absence výčtových datových typů. Řada programátorů obcházela tento nedostatek tak, že místo požadovaného výčtového typu definovala rozhraní a v něm sadu (většinou celočíselných) konstant zastupujících hodnoty definovaného výčtového typu. Chtěl-li někdo tyto konstanty používat, stačilo deklarovat implementaci příslušného rozhraní, a od té chvíle mohla třídy dané hodnoty používat jako by byly její vlastní.

První nevýhoda tohoto přístupu byla koncepční – třída se hlásila k implementaci rozhraní, aniž něco doopravdy implementovala. Navíc se do jejího rozhraní dostaly informace o typech, které používala pouze interně (podrobnosti viz [1]).

Druhou nevýhodou bylo, že tyto konstanty byly většinou definovány jako celočíselné, čímž byla znemožněna jejich typová kontrola. Joshua Bloch v [1] popsal, jak by bylo třeba výčtové typy ve stávající syntaxi definovat. Tato doporučení se stala základem definice výčtových typů ve verzi 5.0.

V definici výčtových typů je klíčové slovo **`class`** nahrazeno klíčovým slovem **`enum`**. Překladač vytvoří třídu, která je potomkem třídy **`java.lang.enum`**, a současně v ní definuje skrytý kód, který později v některých konstrukcích využívá.

Výčtové typy je možno definovat dvěma způsoby. Nejjednodušší definice obsahuje pouze seznam hodnot – např.

```
public enum Období { JARO, LÉTO, PODZIM, ZIMA }
```

V takto jednoduše definované třídě dokonce nemusí být seznam ukončen středníkem. Doporučuji však tuto možnost ignorovat, protože jakmile do těla třídy cokoliv přidáme, musíme přidat i středník ukončující seznam hodnot.

4.1 Překlad do bajtkódu

Jak jsem řekl, překladač doplní do těla třídy definice několika atributů a metod. Podíváme-li se po překladu předchozí definice do útrob vytvořeného souboru .class, nalezneme:

```
public final class Období extends Enum {

    public static final Období JARO;
    public static final Období LETO;
    public static final Období PODZIM;
    public static final Období ZIMA;
    private static final Období[] $VALUES;

    static {
        JARO   = new Období("JARO",   0);
        LETO   = new Období("LETO",   1);
        PODZIM = new Období("PODZIM", 2);
        ZIMA   = new Období("ZIMA",   3);
        $VALUES = new Období[] {JARO, LÉTO, PODZIM, ZIMA};
    }

    public static final Období[] values() {
        return (Období[])($VALUES.clone());
    }

    public static Období valueOf(String name) {
        Období[] arr$ = $VALUES;
        int len$ = arr$.length;
        for(int i$ = 0; i$ < len$; i$++) {
            Období období = arr$[i$];
            if( období.name().equals(name) )
                return období;
            }
        throw new IllegalArgumentException(name);
    }

    private Období(String s, int i) {
        super(s, i);
    }
}
```

Jak vidíte, třída je definována jako potomek třídy **Enum** (přesněji **java.lang.Enum**). Takto však může definovat třídu pouze překladač. Pokus o explicitní deklaraci třídy jako potomka třídy **Enum** je totiž vyhodnocen jako syntaktická chyba.

Překladačem definované metody **values()** a **valueOf(String)** můžete v programu použít, avšak pole **VALUE$** je tak soukromé, že je nelze použít ani v metodách vlastní třídy a potřebujete-li s ním pracovat, musíte si pomocí metody **values()** vytvořit jeho kopii. Překladač se tak brání tomu, aby tvůrci výčtové třídy nahradili prvky pole nějakými jinými.

4.2 Použití hodnot výčtového typu v příkazu switch

Verze 5.0 rozšířila možnosti příkazu **switch** o schopnost použití výrazů, jejichž hodnota je výčtového typu. Hodnoty výčtových typů je pak možno používat i v návěštích **case**. Mohli bychom tedy definovat např. následující metodu:

```
public String příroda( Období období ) {
    switch( období ) {
        case ZIMA:      return "spí";
        case LÉTO:      return "zraje";
        case PODZIM:    return "plodí";
    }
    return null;
}
```

Překladač řeší tyto konstrukce tak, že definuje pomocnou vnořenou třídu, jejímž statickým atributem je vektor mapující ordinální čísla použitých hodnot daného výčtového typu na návěští **case**. Třída obsahující pouze předchozí metodu by se pak přeložila následovně:

```
public class OObdobí {

    static class _cls1 {
        static final int $SwitchMap$Období[];
        static {
            $SwitchMap$Období = new int[Období.values().length];
            try {
                $SwitchMap$Období[Období.ZIMA  .ordinal()] = 1;
                $SwitchMap$Období[Období.LETO  .ordinal()] = 2;
                $SwitchMap$Období[Období.PODZIM.ordinal()] = 3;
            }catch(NoSuchFieldError ex) {}
        }
    }

    public String příroda(Období období)
    {
        switch(_cls1..SwitchMap.Období[období.ordinal()]) {
            case 1: return "spí";
            case 2: return "zraje";
            case 3: return "plodí";
        }
        return null;
    }
}
```

4.3 Definice atributů a metod

V těle výčtového typu můžeme definovat další atributy a metody a můžeme používat i parametrické konstruktory, které překladač interně doplní o dodatečné počáteční dva parametry. Musíme však stále pamatovat na to, že tělo musí začínat seznamem hodnot daného typu.

Upravíme-li definici třídy **`Období`** o možnost pamatovat si aktuální činnost přírody:

```
public enum Období {

    JARO("kvete"), LÉTO("zraje"), PODZIM("plodí"), ZIMA("spí");

    private final String činnost;

    private Období( String činnost ) {
        this.činnost = činnost;
    }

    public String zpráva() {
        return "Je "+name().toLowerCase()+", příroda "+činnost+".";
    }
}
```

bude její přeložený tvar odpovídat definici:

```
public final class Období extends Enum {

   //... Deklarace atributů

    static {
        JARO   = new Období("JARO",   0, "kvete");
        LÉTO   = new Období("LÉTO",   1, "zraje");
        PODZIM = new Období("PODZIM", 2, "plodí");
        ZIMA   = new Období("ZIMA",   3, "spí");
        $VALUES = (new Období[] { JARO, LÉTO, PODZIM, ZIMA });
    }

    private Období(String s, int i, String činnost) {
        super(s, i);
        this.činnost = činnost;
    }

    //... Definice metod
}
```

5 Generické (parametrizované) datové typy

Nejvýznamnějším rozšířením nové verze jazyka je zavedení tzv. generických neboli parametrizovaných datových typů. Protože tyto datové typy nic negenerují, ale pouze využívají typových parametrů k lepšímu instruování překladače, budu v dalším textu dávat přednost termínu parametrizované typy a pro stručnost používat zkratku PDT.

Funkce PDT může někomu připomínat funkci šablon jazyka C++, avšak to je jenom zdání. Šablony C++ a PDT Javy představují dvě různé koncepce, jejichž možnosti se sice částečně překrývají, avšak každá z nich nabízí něco, co ta druhá neposkytuje. Nebudu je zde porovnávat, soustředím se pouze na možnosti PDT.

Hlavním účelem PDT je umožnit typovou kontrolu v situacích, v nichž bylo dříve používáno implicitní přetypování na rodičovskou třídu či implementované rozhraní a posléze explicitní přetypování na vlastní třídu dané instance. Hlavní cíl PDT je přesunout co nejvíce typových kontrol z doby běhu do doby překladu. PDT jsou proto záležitostí překladače. Virtuální stroj se o jejich použití vůbec nedozví. Z přeloženého kódu totiž není možno poznat, jestli původní program PDT používal.

PDT bychom mohli označit jako typy s parametry, jimiž jsou třídy a rozhraní, které budou použity v konkrétní instanci daného parametrizovaného typu. PDT nevytvářejí rodiny typů, jako je tomu u šablon C++. Jedná se pokaždé o týž základní typ, jehož parametry slouží jako dodatečné informace pro překladač, jenž na jejich základě provádí některé dodatečné kontroly a/nebo automaticky vkládá potřebná přetypování.

5.1 Typické použití

Nejčastější použití PDT lze očekávat při práci s dynamickými kontejnery, které doposud neumožňovaly typovou kontrolu vkládaných dat a nutily uživatele k explicitnímu přetypovávání vybíraných dat. Představme si např. následující třídu implementující frontu textových dat.

```
public class Fronta
{
    List<String> názvy = new LinkedList<String>();

    public void zařaď( String název ) {
        názvy.add( název );
    }

    public void zařaď_1( Object objekt ) {
        názvy.add( objekt );
    }

    public void zařaď_2( Object objekt ) {
            String text = (String) objekt;
        názvy.add( text );
    }

    public String další() {
        String ret = názvy.get(0);
        názvy.remove(0);
        return ret;
    }
}
```

V minulých verzích Javy nebylo možno deklarovat typ údajů ukládaných do seznamu **`názvy`**. Překladač proto nemohl zkontrolovat, zda do seznamu ukládáme data povolených typů. Java 5.0 tuto kontrolu díky PDT umožňuje.

Metodu **zařaď(String)** proto nový překladač přeloží bez námitek, avšak metodu **zařaď_1(Object)** přeložit odmítne, protože tato metoda nezaručuje vložení povoleného typu dat. Potřebujeme-li proto opravdu metodu s parametrem typu **Object**, musíme ji upravit např. tak, jak ukazuje metoda **zařaď_2(Object)**. Případná chyba se tentokrát sice objeví stále až v době běhu, ale překladač nás přinutí upravit program tak, aby se projevila již při vkládání dat do kontejneru a na až při jejich vyjímání nebo dokonce až při jejich následném použití.

V metodě **další()** pak překladač automaticky vloží přetypování vyzvedávaného objektu na typ **String**.

Kdybychom chtěli předchozí definici zobecnit a umožnit zadat typ ukládaných dat až při vytváření příslušné fronty, upravili bychom ji následovně:

```
public class Fronta<E>
{
    List<E> názvy = new LinkedList<E>();

    public void zařaď( E název ) {
        názvy.add( název );
    }

    public E další() {
        E ret = názvy.get(0);
        názvy.remove(0);
        return ret;
    }
}
```

Kdybychom pak v programu potřebovali definovat jednu frontu pro texty a druhou pro obrázky, použili bychom příkazy:

```
Fronta<String> texty   = new Fronta<String>();
Fronta<Image>  obrázky = new Fronta<Image>();
```

5.2 Omezení použitelných hodnot typových parametrů

V deklaracích typových parametrů se nemusíme omezovat pouze na prosté zadání použitých datových typů, ale můžeme i blíže specifikovat některé další požadavky na tyto typy, především pak požadované předky nebo implementovaná rozhraní.

Pokud bychom např. chtěli definovat třídu **Interval**, která by reprezentovala interval hodnot, potřebovali bychom zajistit, aby instance datového typu, jehož interval budeme definovat, byly vůbec porovnatelné, tj. aby daný datový typ byl potomkem typu **Comparable**. Potřebná definice by mohla vypadat následovně:

```
public class Interval<T extends Comparable>
{
    private T dolní, horní;

    public Interval( T dolní, T horní ) {
        if( dolní.compareTo( horní ) > 0 )
            throw new IllegalArgumentException(
                "Dolní mez nemůže být větší než horní" );
        this.dolní = dolní;
```

```
        this.horní = horní;
    }

    public T getDolní() { return dolní; }
    public T getHorní() { return horní; }

    public boolean uvnitř( T t ) {
        return (dolní.compareTo( t ) <= 0)  &&
               (t.compareTo( horní ) <= 0);
    }
}
```

V předchozím programu vás možná zarazilo, že u typového parametru **`T`** je uvedeno **`extends Comparable`** místo očekávatelného **`implements Comparable`**. Definice totiž zavádí společné klíčové slovo **`extends`** pro vyjádření dědičnosti tříd i implementace rozhraní.
U typového parametru je možno definovat i požadavek na současnou implementaci několika rozhraní. V takovém případě jsou jednotlivá rozhraní (a případně i rodičovská třída) oddělovány znakem **`&`** – např.:

```
public chytráTřída< T extends Comparable & Serializable >
```

Za použitelný datový typ je přitom považována i třída, jejíhož potomka požadujeme.

5.3 PDT s několika parametry

Čárku není možno k oddělení implementovaných rozhraní použít, protože je vyhrazena pro oddělení jednotlivých typových parametrů v případě, kdy jich zadáváme více.

Pokud bychom se např. rozhodli definovat testovací třídu, v níž bychom chtěli demonstrovat rozdílnou výkonnost různých implementací rozhraní **`List`**, mohli bychom ji definovat následovně:

```
public class Seznamy < L extends List<E>, E >
{
    private static List<E> seznam;
    private static E element;

    public Seznamy( L seznam, E element ) {
        this.seznam  = seznam;
        this.element = element;
    }

    // ... Testovací metody

    public static void test() {
        Seznamy s;
        s = new Seznamy< ArrayList<String>, String >
                ( new ArrayList<String>(), "ArrayList" );
        // ... sada testů s danou instancí
        s = new Seznamy< Vector<String>, String >
                ( new Vector<String>(), "Vektor" );
        // ... sada testů s danou instancí
    }
}
```

5.4 PDT v definicích metod

PDT je možno využít nejenom při definici třídy, ale i při definici jednotlivých metod v třídách, které PDT samy nevyužívají. Pokud bychom např. potřebovali definovat metodu, která zatřídí zadaný prvek do zadaného seznamu podle velikosti, mohla by její definice vypadat následovně:

```
public <E extends Comparable> void zatřiď( E element, List<E> sada ) {
    ListIterator<E> li = sada.listIterator();
    while( li.hasNext() ) {
        if( element.copareTo( li.next() ) >= 0 )
        {
            li.add( element );
            return;     //==========>
        }
    }
    li.add( element );
}
```

5.5 Ignorování dědičnosti typových parametrů

Při práci s PDT musíme dát pozor na to, abychom na ně nepřenesli naše zkušenosti z práce s poli a dynamickými kontejnery. Mezi instancemi třídy 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;
/* 3 */  obj.add(new Object());
/* 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*.

5.6 Zástupné typy jako typové parametry

Dva PDT, které se liší pouze hodnotami svých typových parametrů, jsou považovány vždy za vzájemně nezávislé bez ohledu na to, jsou-li jejich typové parametry nějak příbuzensky spřízněny. Tato skutečnost však při používání PDT velmi svazuje ruce. Když jsme ve starší verzi Javy definovali metodu

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

mohli jsme s její pomocí tisknout obsah 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 PDT nesvazovaly ruce svojí neschopností zavést vztahy ekvivalentní dědičnosti, zavádí Java 5.0 *zástupný typ* (wildcard type) ? (otazník), který označuje libovolný typ vyhovující případným omezujícím podmínkám.

K PDT používajícímu zástupný typ se pak můžeme chovat obdobně jako k rodiči PDT, jenž za zástupný typ dosadil nějaký konkrétní typ a naopak ke PDT s konkrétním typovým parametrem se můžeme chovat jako k potomkovi odpovídajícího PDT se zástupným typem.

Kdybychom použili v našem předchozím příkladu zástupný typ, 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.

5.7 Omezení použitelných hodnot zástupných typových parametrů

I na zástupné typy můžeme klást dodatečná omezení. Představte si např. sadu tříd představujících grafické objekty, které jsou schopny se nakreslit na virtuální plátno. Všechny třídy těchto objektů budou implementovat rozhraní **`IKreslený`** vyžadující implementaci metody **`nakresli()`**.

Kdybychom chtěli definovat metodu, která je schopna nechat nakreslit všechny prvky v zadané kolekci, nemůžeme použít metodu deklarovanou následovně:

```
public void nakresli( Collection<IKreslený> c )
```

Jejím parametrem totiž může být pouze kolekce, která za své prvky deklarovala instance rozhraní **`IKreslený`**. Nemůže jím ale být kolekce instancí kterékoliv z tříd, jež toto rozhraní implementují, ani kolekce případných potomků tohoto rozhraní.

Pokud bychom chtěli deklarovat metodu, která bude jako svůj parametr akceptovat kolekci čehokoliv, co je schopno se nakreslit (přesněji čehokoliv, co implementuje rozhraní **`IKreslený`**), musíme ji deklarovat

```
public void nakresli( Collection<? extends IKreslený> c )
```

Zástupné typy můžeme použít nejenom při deklaraci parametrů metod, ale i při deklaraci typu proměnných a atributů. Metoda testující právě deklarovanou metodu by pak mohla vypadat následovně:

```
public void test()
{
    List<? extends IKreslený> lik;
    lik = new ArrayList<Tvar>();   naplň( lik );   nakresli( lik );
    lik = new LinkedList<Kruh>();  naplň( lik );   nakresli( lik );
}
```

5.8 Omezení PDT

Při používání PDT a jejich typových parametrů musíme respektovat řadu omezení, která vyplývají většinou z toho, že nám při jejich používání některé informace o použitém typu chybí.

Pouze objektové typy. Jak z předchozího textu vyplývá, použití PDT je omezeno pouze na objektové typy. Jako typový parametr proto není možno uvést žádný z primitivních typů.

Ztráta informace při běhu. Jak jsme již řekli na počátku pasáže o PDT, veškeré informace ve zdrojovém kódu jsou určeny pro překladač. Při běhu programu je již není možné použít. Různé testy, které se vyhodnocují až za běhu programu, proto nemohou brát tyto informace v úvahu. Např. podmínka

```
seznam instaceof List<String>
```

může otestovat pouze to, je-li proměnná **seznam** instancí rozhraní **List**. O tom, co má daný seznam obsahovat, se při běhu již nic nedozvíte.

Obdobně i operátor přetypování může přetypovat svůj operand pouze na základní typ, avšak nemůže již zabezpečit použití dodatečné informace. Příkaz

```
List<String> seznam1 = (List<String>) seznam2;
```

bude pracovat naprosto stejně, jako kdybyste místo parametrizovaných typů použili pouze základní typ **List**.

Výjimky. PDT nemohou být potomky typu **Throwable**. Program proto nemůže vyhodit výjimku parametrizovaného typu ani výjimku takovéhoto typu zachytávat.

Současně není možno použít typový parametr ke specifikaci zachytávané výjimky v klauzuli **catch**. Typový parametr je však možno použít k definici typu vyhazované výjimky. Lze tedy definovat následující metodu:

```
public <T extends Throwable> void zkus( T t ) throws T {
    try {   //Zkus provést požadovanou činnost
    }catch( Throwable výjimka ) {
        t.initCause( výjimka );
        throw t;
    }
}
```

Pole. Není možné vytvářet pole s prvky parametrizovaného typu. Pokus o jeho vytvoření vyvolá syntaktickou chybu. Není tedy možno zadat:

```
List<String>[] seznam = new ArrayList<String>[4];
```

Je však možno deklarovat proměnnou, která je polem PDT, a inicializovat ji odkazem na instanci neparametrizovaného typu. Předchozí příklad lze tedy upravit:

```
List<String>[] seznam = new ArrayList[4];
```

Tento příkaz sice vyvolá při překladu varování, nicméně se v pořádku přeloží a proměnnou **`seznam`** je pak možno používat v souladu s její deklarací.

A další … Obdobných omezení je ještě celá řada, avšak rozsah příspěvku je nedovoluje všechny vyjmenovat a vysvětlit. Musím vás proto odkázat na literaturu, která se pomalu začíná objevovat a v blízké budoucnosti jí bude jistě dostatek.

6 Rozšíření příkazu for

Při práci s kontejnery potřebujeme často provést nějakou operaci se všemi objekty uloženými v daném kontejneru. Řada programovacích jazyků definuje pro tento účel konstrukci nazývanou obecně **`for-each`**. Obdobnou konstrukci zavedla i Java. Protože však autoři nechtěli zvyšovat počet klíčových slov, zavedli konstrukci následovně:

`for(` *typ-prvků-kontejneru parametr-cyklu* **`:`** *kontejner* **`)`** *tělo-cyklu*

Uvedený příkaz lze přitom použít na všechny typy kontejnerů, tj. jak na dynamické kontejnery, tak na klasická pole. Pokud bychom např. chtěli definovat metodu, která by vrátila průměr hodnot uložených v poli, mohli bychom ji definovat takto:

```
public double průměr( double[] da ) {
    double s = 0;
    for( double d : da )
        s += d;
    return s / da.length;
}
```

Prakticky naprosto stejně bychom metodu definovali i v případě, kdy by byly hodnoty místo v poli uloženy v nějakém dynamickém kontejneru (kolekci). V takovém případě bychom mohli navíc využít PDT a tím tak zabezpečit, že případný pokus o zavolání metody pro kolekci neobsahující data, jež je možno převádět na čísla typu **`double`**, vyvolá chybu překladu.

```
public <T extends Number> double průměr( Collection<T> sada ) {
    double součet = 0;
    for( T t : sada )
        součet += t.doubleValue();
    return (součet / sada.size());
}
```

7 Metody s proměnným počtem parametrů

Dalším drobným rozšířením nové verze jazyka je povolení metod s proměnným počtem parametrů. Přiznejme si, že toto rozšíření je opět pouze kosmetické, protože tyto metody jsou funkčně naprosto ekvivalentní metodám, jejichž posledním parametrem je vektor (jednorozměrné pole). Jediným rozdílem je, že definuji-li poslední parametr v hlavičce jako variantní, nemusím při volání metody vytvářet explicitně potřebný vektor, ale mohu pouze napsat seznam příslušných parametrů a překladač vytvoří příslušný vektor za mne.

Proměnný počet parametrů deklarujeme v hlavičce metody tak, že mezi typ těchto parametrů a název pole, v němž budou metodě předány, napíšeme tří tečky. Metodu, která bude vracet průměr zadaných hodnot, bychom tedy mohli definovat následovně:

```
public static double průměr( double ... dd ) {
    double součet = 0;
    for( double d : dd )   součet += d;
    return součet / dd.length;
}
```

Metodu pak můžeme v programu volat následovně:

```
průměr( 1, 2, 3, 4 );
průměr();
```

V druhém případě vrátí metoda hodnotu **NaN** jako výsledek dělení nuly nulou.

8 Podpora znaků podle specifikace Unicode 4.0

Dosavadní verze Javy byly schopné pracovat pouze s dvoubajtovými znaky v rozsahu '\u0000' až '\uFFFF'. Od vzniku Javy se však množina znaků definovaných ve specifikaci Unicode rozrostla a místo 16bitového kódování používá 21bitové, tj. zahrnuje znaky s kódem od 0x0 do 0x10FFFF.

Nová verze Javy podporuje práci s úplnou množinou těchto znaků, i když poněkud nestandardním způsobem. Znakový typ **char** zůstává z důvodu kompatibility i nadále 16bitový a všechny metody, jejichž parametr má typ **char**, mohou i nadále zpracovávat pouze původní sadu UTF-16. S kompletní sadou znaků mohou pracovat pouze metody, které sice podle kontraktu očekávají v parametru kód znaku, avšak používají parametr typu **int**.

Některé třídy, především pak třída **Character**, byly v nové verzi doplněny o rozšiřující sadu metod, které jsou ekvivalentní stávající metodám, avšak místo parametru typu **char** používají parametr typu **int**, resp. metodami, které místo hodnoty typu **char** vracejí hodnotu typu **int**.

Znaky, které nespadají do základní sady, se v řetězcích zadávají jako dvojice znaků, přičemž kód prvního musí být v rozsahu 0xD800 až 0xDBFF a kód druhého musí být v rozsahu 0xDC00 až 0xDFFF, např.:

```
String u="Znak [\uD840\uDC08]";
```

9 Metadata

Stále častěji se objevují nástroje a technologie, které ke své práci vyžadují informace, jež není možno jednoduše zanést do vlastního kódu. Např. mnohá API, zejména pak API pro distribuované systémy a webové služby, požadují po programátorech, aby vedle vlastního kódu vytvářené komponenty naprogramovali ještě množství různé „vaty“ potřebné pro správnou funkci výsledného programu (většinou to jsou nějaká podpůrná rozhraní). U této „vaty“ se předem přesně ví, jak má vypadat, takže její příprava je v podstatě mechanickou záležitostí. Java 5.0 přichází s myšlenkou odbourání této mechanické práce zavedením tzv. metadat.

V současné době bychom např. nejspíš definovali komponentu pro správu objednávek zhruba následovně:

```
public class ObjednávkyImpl implements Objednávky {

    public Zboží[] getNabídka() {
        //...
    }
    public String objednej(String název, int počet) {
        //...
    }
}
```

Budeme-li chtít použít tuto komponentu jako EJB, vyžaduje J2EE pro správnou činnost této komponenty definici rozhraní („vata“):

```
public interface Objednávky extends java.rmi.Remote {

    public Zboží[] getNabídka()
        throws java.rmi.RemoteException;

    public String objednej(String název, int počet)
        throws java.rmi.RemoteException;
}
```

Klíčovou povinností vývojáře je průběžně sledovat, aby se při všech modifikacích promítly změny kódu do oné „vaty“ a naopak. Tvůrci Javy se inspirovali atributy jazyka C# a zavedli do jazyka tzv. metadata, která překladač zpracuje a uloží do přeložených souborů jako dodatečné informace, jež nejsou přímou součástí kódu určeného k interpretaci. Kdybychom pracovali ve vývojovém prostředí, které umí využít výhod metadat, mohli bychom kód komponenty definovat např. následovně:

```
public class ObjednávkyImpl implements Objednávky {

    @Remote public Zboží[] getNabídka() {
        //...
    }
    @Remote public String objednej(String název, int počet) {
        //...
    }
}
```

Metadatový příkaz `@Remote` by pak zabezpečil, aby použité vývojové prostředí vygenerovalo potřebné rozhraní při překladu a sestavování aplikace samo.

Možnosti využití metadat jsou mnohem širší a lze očekávat, že mnohé z nich budou teprve objeveny. Máme se tedy v nejbližších letech na co těšit.

Reference

1. Bloch J.: *Effective Java – Programming Language Guide*, Addison Wesley, 2001. Český překlad *Java efektivně – 57 zásad softwarového experta*, Grada 2002, ISBN 80-247-0416-1.

Annotation

The New Features of Java 5.0

The last version of Java Language brings the most fundamental changes in its history. This time not only the standard library was updated and extended, but changes significantly touched the syntax, too. The paper presents the syntactic news the Java 5.0 come with. It stepwise discuss the static import, autoboxing and unboxing, object enumerated types, generic (parameterized) types, extension of the for statement, methods with variable number of parameters and annotations (metadata).