J2SE 1.5 Tiger JAVA

Bevezetés

2004 februárjában az Early Access keretében vált letölthetővé a J2SE következő verziója(**http://java. Sun. com/j2se /1.5.0/download.jsp**), melynek verziószáma 1.5.0 Beta 1. Ennek megjelenését hatalmas figyelem kísérte. Már régóta lehetett hallani pletykákat és hivatalos nyilatkozatokat arról, hogy mit is fog tartalmazni, milyen újításokat fog nyújtani.

A Java fejlesztők három fő platformot választhatnak, ha alkalmazás írásába szeretnének fogni. A J2SE egy komplett alap környezet kliens-szerver és asztali alkalmazások készítéséhez; a J2EE nagyvállalati környezetben használható technológiákat tartalmaz; míg a J2ME alkalmazási területe a mobil fejlesztés.

Most tehát az alap platformból jött ki új verzió, mely egyrészt jelentős nyelvi újdonságokat is tartalmaz, másrészt a már meglévő API-k bővültek, új API-k kerültek bele, sőt jelentős mennyiségű hibajavításon és optimalizáción is átesett. Az új verzió fejlesztésekor főleg négy területet tartottak szem előtt: a fejlesztés kényelmét, monitorozást és menedzselhetőséget, skálázhatóságot és teljesítményt, valamint XML és kliens oldali Web szolgáltatások támogatását.

A J2SE 1.5 jelenleg 15 új Java Specification Request - JSR specifikációt implementál, és közel 100 nagyobb frissítést tartalmaz, melyeket a Java Community Process (JCP) keretein belül fejlesztettek.
A négy fő fejlesztési terület közül az egyik a fejlesztés könnyítésére irányul (Ease of Development), melyet a következő nyelvi elemek és tulajdonságok hivatottak biztosítani: generikus típusok (JSR 14), metaadatok (JSR 175), automatikus be- és kicsomagolás, fejlettebb ciklusképzés, felsorolásos típus, statikus import (ez utóbbi négyet a JSR 201 tartalmazza), C típusú formázott adatbevitel és kiírás, változó számú paraméterlista, párhuzamos programozást segítő eszközök és az RMI interfész generálás elhagyása.
Alapértelmezésben a fordító 1.4-es Java kódot fordít, a **source** 1.5 kapcsolóval lehet megadni, hogy az 1.5 lehetőségeit kihasználjuk.

Metaadatok

A metaadatokat a Java forrás szövegében helyezhetjük el, osztályokhoz, interfészekhez, metódusokhoz és mezőkhöz kötve. Ezek olyan kiegészítő információk, melyeket feldolgozhat a Java fordító, bármilyen automatizált eszköz, de akár a bájtkódba is bekerülhet, és Java Reflection API-val elérhetővé válik.

A metaadat egy olyan eszköz, mely kiváló infrastruktúrát biztosíthat különböző kisegítő vagy deploy fájlok, kötelező interfészek generálásra, esetleg naplózás könnyítésére.

Generikus típus

A generikus típus bevezetését a Java fejlesztők már hosszú ideje várták. A generikusok hasonlítanak a C templatekre, de alapvető különbségek is vannak. A generikusok a típusoknál még egyel magasabb absztrakciós szinten helyezkednek el. Osztályok, interfészek és metódusok paraméterezhetők típusokkal. A fejlesztő legelőször a Collection API használata közben találkozhat a generikus típusokkal.

A Collection API osztályai képesek bármilyen osztályú objektumok kezelésére, de legtöbbször arra van szükségünk, hogy egy ilyen Collection API-hoz tartozó osztály csak egy osztályhoz tartozó objektumokat kezeljen. Abban az esetben, ha egy ilyen Collection API-hoz tartozó osztályba eltérő objektumokat teszünk bele, akkor az nem derül ki fordításidőben, csak futásidőben, és ilyenkor egy **ClassCastException** kivételt kaphatunk, ha egy elemet ki akarunk venni.

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue(); /* Itt váltódhat
ki a kivétel, ha nem Integer-ré típuskényszerítünk. */
```

Ugyanez egy olyan **ArrayList** osztállyal, mely a generikuson alapul:

```
ArrayList<Integer> list =  new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Az **ArrayList** egy generikus interfész, ami egy típus paramétert vár.

Ebben az esetben, ha az **add** metódust egy olyan paraméterrel hívnánk meg, mely nem az **Integer** osztály egy példánya, már fordítási időben hibát kapnánk. Ezen kívül egy elem kivételekor sincs szükségünk típuskényszerítésre.

Tehát a generikusok használatával lehetőségünk van általános objektumokat kezelő osztályok készítésére, melynél megadható használatkor, hogy konkrétan milyen osztályú objektumokat kezeljen, növelve így a biztonságot és olvashatóságot, kiküszöbölve a felesleges típuskényszerítést, zárójeleket és az ideiglenes változókat. Ha olyan metódusokat írunk, melyeket mások is használnak, eddig csak JavaDoc megjegyzésben tudtuk megadni, hogy a paraméterként kapott Collection milyen osztályú objektumokat tartalmazzon. A generikusok használatával csak olyan Collectiont lehet átadni, melyre a megírt metódus számít. Ez a metódus szignatúrájából látszik, és a hiba már fordítási időben kiderül.

Az előző példa egy már implementált generikus típusú osztályt mutatott be, de lehetőség van arra, hogy saját magunk is készítsünk ilyent, alapul véve például a Collection API forráskódját.
Részlet:

```
interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}

interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

A relációs jelek között a formális paraméter típust adjuk meg. A típus paramétereket legtöbbször ott használjuk, ahol a szokványos típusokat is.

Amikor meghívjuk a generikus deklarációját, akkor a formális paraméter típust a futtató környezet kicseréli a hívásnál megadott típussal. Ez a csere a valóságban azonban nem jelenik meg, nem lesz új entitás sem forrás, sem bájtkód szinten, sem a lemezen vagy memóriában, csak egyszer lesz lefordítva, egyetlen egy class fájllá.
A formális paraméter típus neve legyen minél rövidebb, lehetőleg egy karakterből álló, és első karaktere legyen nagybetű. Ahogy a példa is mutatja a Collection API formális paraméter típusának neve E.

Fontos tulajdonsága a generikusoknak, hogy ha egy generikusnak egy paramétert átadunk, és ugyanazon generikusnak ezen paraméter egy altípusát (alosztály vagy alinterfész), akkor az altípus reláció nem marad meg a generikusoknál. Például:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

Az ls nem altípusa az lo-nak, így a fordító a második sorra hibát fog jelezni. Hiszen nézzük a következő példát:

```
lo.add(new Object());
String s = ls.get(0);
```

Ebben az esetben a második sor egy **ClassCastException** hibát dobna, így megszűnne a generikusok típus biztonsága.

Ilyen módon tehát az sem lehetséges, hogy olyan metódust írjunk, mely egy **Collection<Object>** típust kap paraméterként, és kezeli az összes más paraméterrel szereplő generikust, hiszen a **Collection<Object>** nem „őse” például a **Collection<String>** generikusnak. E problémára találták ki a wildcard típust, így lehet például egy „ismeretlen típusú Collectiont” paraméterként kapó metódust deklarálni:

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}
```

Az előzőekből kifolyólag a következő kódrészlet második sora szintén fordítási hibát adna, és ez a bizonyíték arra, hogy a típus biztonság garantált:

```
Collection<?> c = new ArrayList<String>;
c.add(new Object());
```

Az **add** paraméterként egy E típust várna, de mivel a c deklarációja ezt a típust nem határozza meg, ezért a fordító nem tudja eldönteni, hogy az **Object** osztályú példány megfelelő-e a biztonság szempontjából.

A **get** metódust viszont tudjuk használni, hiszen az mindenképpen egy Object osztály vagy leszármazottjának egy példányát fogja visszaadni, hiszen minden Java osztálynak az **Object** osztály az őse.

Az osztályok öröklődését azonban mégis felhasználhatjuk a generikusoknál, méghozzá a következőképpen:

```
public void drawAll(List<? extends Shape> shapes) {... }
```

Ebben az esetben viszont a **drawAll** metódusnak paraméterként adhatunk olyan **List**-et is, melynek aktuális paraméter típusa a **Shape** osztály valamely leszármazottja. Ennek a deklarációnak a neve korlátozott wildcard (bounded wildcard), ahol a paraméter típusa szintén ismeretlen, csak annyit tudunk róla, hogy a **Shape** osztály vagy annak valamely leszármazottja. Ezt úgy mondjuk, hogy a **Shape** a wildcard felső korlátja (upper bound). A következő sor hibás lenne az előzőleg definiált metódusban:

```
shapes.add(0, new Rectangle());
```

A **Map** is genrikus típus, melynek két paraméter típusa van, név konvenció szerint K és V, azaz „Key” és „Value”.
Az előbbiek miatt például a következő metódus sem megfelelő:

```
static void fromArrayToCollection(Object[] a,
Collection<?> c) {
  for (Object o : a) {
    c.add(o);
  }
}
```

A második sor fordítási hibát eredményez. A kívánt funkció generikus metódusokkal azonban mégis megoldható, ekkor a metódust is paraméterezhetjük különböző típusokkal:

```
<T> static void fromArrayToCollection(T[] a,
Collection<T> c) {
  for (T o : a) {
    c.add(o);
  }
}
```

A metódus hívásakor nem kell paraméter típust átadni, ehelyett a fordító automatikusan meghatározza az aktuális paraméter típusát véve alapul.

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>;
fromArrayToCollection(oa, co);
```

A generikusoknak több felhasználási módja is lehetséges, melyekre a cikk már nem terjed ki, ehelyett csak megemlíti ezeket, a részletek kikeresése már az Olvasó feladata.

A generikusok fejlesztői külön figyelmet fordítottak arra, hogy a régi kódok, illetve az új, generikusokra épülő kódok jól megférjenek egymás mellett, és fokozatosan lehessen a régi kódot migrálni.

Érdemes megjegyezni, hogy egy generikus osztálya független a paraméterként megadott típustól, hiszen csak egy osztály tartozik minden generikushoz, bárhogy is paraméterezzük. Erre kell figyelni az instanceof operátor használatakor, valamint típuskényszerítésnél is.
Egy tömb elemeinek típusa nem lehet paraméterezett generikus, helyette wildcard tömböket lehet használni.

A **java.lang.Class** osztály is generikus.
Létezik wildcard alsó korláttal (lower bound), melynek kulcsszava a super.

Automatikus be- és kicsomagolás

A primitív típusok kezelése a Java nyelvben bizonyos esetekben igen körülményes. Mivel egy Collection-be csak az Object osztályból leszármazó osztályok objektumait lehet beilleszteni, a primitív típusokat át kellett konvertálni osztály szintű megfelelőikbe. A primitív típusok automatikus be- és kicsomagolása (autoboxing és auto-unboxing) lehetőséget ad arra, hogy ne kelljen a primitív típusokat osztály szintű megfelelőikbe (wrapper) becsomagolnunk, hanem ez automatikusan történjen. Az előző példát folytatva a kód a következőre egyszerűsödik:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Abban az esetben, ha például egy **Hastable** példányból szeretnénk egy **Integer** típusú értéket automatikus kicsomagolással int változó értékének kinyerni, de a kapott **Integer** objektum **null** lenne, **NullPointerException** kivételt kapunk:

```
Map<String, Integer> map = new Hashtable<String,
Integer>();
map.put("one", 1);
int two = map.get("two"); // NullPointerException
```

Fejlettebb ciklusképzés

A fejlettebb ciklusképzés (foreach) lehetőséget biztosít az **Iterator** osztály körülményes használatának és a típuskényszerítésnek a kikerülésére a Collection API használata esetén.
A kód e tulajdonság használata nélkül:

```
ArrayList list = new ArrayList();
for (Iterator i = list.iterator(); i.hasNext();) {
  Integer value=(Integer)i.next();
}
```

Az új ciklus használatával:

```
ArrayList list = new ArrayList();
for (Object o: list) {
  Integer i = (Integer) o;
}
```

Generikus használatával a kód még egyszerűbbé válik:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Integer i : list) {
}
```

A tömbökre is működik:

```
int[] array = new int[10];
for (int i: array) {
}
```

Felsorolásos típus

A felsorolásos típus is most jelent meg a Java nyelvben. Ez eddig is egy gyakran használt tervezési minta volt, de most nyelvi szinten elérhetővé vált a következő szintaktikával:

```
public enum StopLight {red, amber, green };
```

A felsorolásos típus előnyei az int típusú konstansokkal szemben:

- Fordítási időben történik az ellenőrzés;
- A konstansoknak egy névteret biztosít;
- Hibakeresés esetén a kiírás sokkal informatívabb jellegű;
- Hatékonyan alkalmazhatóak a **switch** kifejezésben;
- Mivel ez is objektum, használható a Collection API osztályaiban;
- Mivel ez alapvetően egy osztály, mezők és metódusok adhatók hozzá.

Egy összetettebb példa, mely az utolsó előnyre mutat példát:

```
public enum Coin {
   penny(1), nickel(5), dime(10), quarter(25);
   Coin(int value) {this.value = value; }
   private final int value;
   public int value() {return value; }
}
```

Statikus import

A statikus import lehetőséget ad arra, hogy egy osztály statikus mezőire úgy kelljen hivatkozni, hogy meg kell adni az osztály nevét. A statikus import szintaktikája:

```
import static java.util.Calendar.*;
Calendar c = new Calendar();
C.get(DAY_OF_MONTH);
```

Formázott adatbevitel és kiírás

Megjelent a C típusú formázott adatbevitel és kiírás lehetősége is, melyet a C nyelvben a **scanf** és **printf** függvény valósított meg. Java nyelvi megfelelői:

```
System.out.printf("%s %5d\n", user,total);
Scanner s = Scanner.create(System.in);
String param = s.next();
int value = s.nextInt();
S.close();
```

Ezzel kapcsolatban több információt a **java.util.Formatter** osztály API dokumentációja ad.

Változó számú paraméterlista

A változó számú paraméterlistát is a C nyelvből vették át a következő szintaxissal:

```
void argtest(Object ... args) {
 for (int i=0;i <args.length; i++) {
   System.out.println(args[i]);
 }
}
Argtest("test", "data");
```

A példa **argtest** metódusát bármilyen számú és típusú paraméterrel meg lehet hívni, melyeket a metódus kiír.

Párhuzamos programok

A párhuzamos programok fejlesztésére is jelentős számú új funkcionalitást vezettek be, melyek a JSR-166-on alapulnak.

Dinamikus proxyk

Nincs többé szükség az rmic (RMI compiler) eszközre, ami a távoli interfész vázakat generálta, ehelyett a dinamikus proxy-k bevezetésével a régebben a vázak által biztosított adatok futásidőben felderíthetők.

Konklúzió

A Java 1.5-ben megjelenő újdonságokra a Java programozók már régóta vártak. A tervezők maximálisan figyelembe vették a visszafelé kompatibilitást, a régi programok migrációjának lehetőségét. Az új funkciók segítik a programozást, használatukkal átláthatóbb, tisztább, és biztonságosabb programokat lehet írni.

Az átállásra még van idejük a fejlesztőknek, hiszen a közelmúltban jelent meg az Early Access program keretében a beta verzió, és alig van olyan fejlesztőeszköz, mely támogatná az új lehetőségeket. De addig is érdemes velük megismerkedni, hogy könnyen váltani tudjunk.
Ez a cikk csak néhány új nyelvi elemet mutat be nagy vonalakban, de ezeken kívül rengeteg új funkció, újítás van az új verzióban, melyek jelentősen könnyíthetik a programozó munkáját, így megismerésük feltétlenül ajánlott.

Hivatkozások

A szerzőről

Viczián István a Debreceni Egyetem program-tervező matematikus szakán végzett 2001 nyarán, most ugyanott levelező PhD hallgató az elosztott rendszerek, middleware-ek témakörében. Jelenleg vezető fejlesztőként dolgozik Budapesten, melynek keretében csoportmunkát segítő eszközökkel, Java nyelvvel, webes technológiákkal, middleware szoftverekkel és alkalmazásintegrációval foglalkozik. Szabadidejében optimalizációs alkalmazást fejleszt, új Java technológiákkal ismerkedik, valamint Java blog-ot ír (JTechLog).

e-mail: viczus@freemail.hu
Honlap: http://dragon.unideb.hu/~vicziani