Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Generikus típusok

Sipos Róbert
siposr@hit.bme.hu

2014. 02. 27.

Bevezető példa I.

```
public interface List {
   public void add (Object value);
   public ListIterator iterator ();
}
```

```
public class LinkedList implements List {
   public void add (Object value) {
      ListElement listElement = new ListElement (value);
      if (root == null)
         root = listElement;
      else {
         listElement.setNext (root);
         root = listElement;
      }
   }

   public ListIterator iterator () {
      return new ListIterator (root);
   }

   private ListElement root;
}
```

Bevezető példa II.

```
public class ListElement {

   public ListElement (Object value) {
      this.value = value;
   }

   public Object getValue () {
      return value;
   }

   public ListElement getNext () {
      return next;
   }

   public void setNext (ListElement next) {
      this.next = next;
   }

   private Object      value;
   private ListElement next;
}
```

Bevezető példa III.

```
public class ListIterator {

   public ListIterator (ListElement listElement) {
      this.listElement = listElement;
   }

   public Object next () {
      if (listElement == null)
         return null;
      ListElement next = listElement;
      listElement = listElement.getNext ();
      return next.getValue ();
   }

   public boolean hasNext () {
      return listElement != null;
   }

   private ListElement listElement;
}
```

Bevezető példa IV.

```
public class ListTest {

   public static void main (String[] args) {
      List l = new LinkedList ();
      l.add ("alma");
      l.add ("korte");
      l.add ("szilva");
      for (ListIterator i = l.iterator ();i.hasNext ();)
         System.out.println (i.next ());
   }
}
```

- A fenti példában nem törődünk vele, hogy a listába milyen típusú objektumokat tettünk.

Bevezető példa V.

```
public class ListTest {

   public static void main (String[] args) {
      List l = new LinkedList ();
      l.add ("alma");                                  // ClassCastException
      l.add (new Integer (3));
      l.add (new Integer (4));
      for (ListIterator i = l.iterator ();i.hasNext ();)
         System.out.println (((Integer) i.next ()).intValue () + 1);
   }
}
```

- Most viszont baj van, mert az "alma" miatt ClassCastException dobódik.

Bevezetés I.

- A Java típusossága csak "elsőrendű", a típusokat nem lehet tovább specifikálni, legfeljebb leszármaztatással.
- Ez baj, mert ha általános célú típusokat (osztályokat, interfészeket) definiálunk, minden paraméter és visszatérési érték csak a lehető legáltalánosabb típusú lehet (tipikusan Object).
- Ebből következően **csak futásidejű típusellenőrzés van**, a kódot pedig olvashatatlanná teszik az **explicit típuskonverziók** (castok).

Bevezetés II.

- A J2SE 1.5-ös verziója nyelvi támogatást nyújt a fenti problémákra, ez a **generics**.
- A generics összefoglaló név ún. **generikus típus**ok (generic type) és **generikus metódus**ok (generic method) definiálásának lehetőségét jelenti.
- A generikus típus és metódus **típussal paraméterezhető** típust és metódust jelent.

Paraméterezett típusok

- A generikus típusokból **típusparaméter**ek megadásával **paraméterezett típus**ok (parameterized type) hozhatók létre:
- A paraméterezett típusok mindenütt használhatók, ahol szokványos típusok, tehát más paraméterezett típusok típusparamétereiként is.
- Primitív típusok filozófiai és technikai okokból sem használhatók típusparaméterekként.

Paraméterezett típusdeklarációk I.

- Egy generikus osztály, illetve interfész deklaráció **típusok egy halmazát definiálja**.
- A halmaz egyes elemeit a megadott típusparaméterek határozzák meg.

```
class Pair<N,M> {
   public N getFirst () { ... };
   public M getSecond () { ... };
}
```

- A **<** és **>** jelek között **típusváltozó**k szerepelnek.
- A típusváltozókat a deklaráció további részében típusként használhatjuk.

Példa I.

```
public interface List<E> {
   public void add (E value);
   public ListIterator<E> iterator ();
}
```

```
public class LinkedList<E> implements List<E> {
   public void add (E value) {
      ListElement<E> listElement = new ListElement<E> (value);
      if (root == null)
         root = listElement;
      else {
         listElement.setNext (root);
         root = listElement;
      }
   }

   public ListIterator<E> iterator () {
      return new ListIterator<E> (root);
   }

   private ListElement<E> root;
}
```

Példa II.

```
public class ListElement<E> {

   public ListElement (E value) {
      this.value = value;
   }

   public E getValue () {
      return value;
   }

   public ListElement<E> getNext () {
      return next;
   }

   public void setNext (ListElement<E> next) {
      this.next = next;
   }

   private E               value;
   private ListElement<E> next;
}
```

Példa III.

```
public class ListIterator<E> {

   public ListIterator (ListElement<E> listElement) {
      this.listElement = listElement;
   }

   public E next () {
      if (listElement == null)
         return null;
      ListElement<E> next = listElement;
      listElement = listElement.getNext ();
      return next.getValue ();
   }

   public boolean hasNext () {
      return listElement != null;
   }

   private ListElement<E> listElement;
}
```

Példa IV.

```
public class ListTest {

   public static void main (String[] args) {
      List<String> l = new LinkedList<String> ();
      l.add ("alma");
      l.add ("korte");
      l.add ("szilva");
      for (ListIterator<String> i = l.iterator ();i.hasNext ();)
         System.out.println (i.next ());
   }
}
```

- Noha nem használjuk ki azt, hogy String-eket tárolunk a listában, jelöljük, így fordításidejű típusellenőrzést kapunk.

Példa V.

```
public class ListTest {

   public static void main (String[] args) {
      List<Integer> l = new LinkedList<Integer> ();
      l.add ("alma");                                    // Fordítási hiba
      l.add (new Integer (3));
      l.add (new Integer (4));
      for (ListIterator<Integer> i = l.iterator ();i.hasNext ();)
         System.out.println (i.next ().intValue () + 1);
   }
}
```

- A bekeveredett String miatt most **fordítási hiba keletkezik, nem futásidejű**.
- A képet tovább egyszerűsíthetjük az autoboxing/unboxing használatával.

Példa VI.

```
public class ListTest {

   public static void main (String[] args) {
      List<Integer> l = new LinkedList<Integer> ();
      l.add (3);                                          // autoboxing
      l.add (4);                                          // autoboxing
      for (ListIterator<Integer> i = l.iterator ();i.hasNext ();)
         System.out.println (i.next () + 1);           // autounboxing
   }
}
```

Példa VII.

```
public class ListTest {

   public static void main (String[] args) {
      List<Integer> l = new LinkedList<> (); // Java7-től, “diamond”
      l.add (3);
      l.add (4);
      for (ListIterator<Integer> i = l.iterator ();i.hasNext ();)
         System.out.println (i.next () + 1);
   }
}
```

Paraméterezett típusdeklarációk II.

- A típusváltozókhoz az **extends** kulcsszóval korlátokat is megadhatunk, amelyek a behelyettesíthető típusparaméterek körét szűkítik.
- Korlátként megadhatunk egy osztályt vagy interfészt, illetve **&** jelekkel elválasztva tetszőleges számú további interfészt.

```
class Pair<N extends Number&Comparable,M extends Iterator> {
    ...
}
```

- A megadott korlátok teljesülését a fordító ellenőrzi:

```
new Pair<Integer,ListIterator> ( ... )                    // OK
new Pair<LinkedList,Iterator> ( ... )          // fordítási hiba
```

Paraméterezett típusdeklarációk III.

- A korlátok célja a generikus típus paraméterezhetőségének szabályozása.
- A korlátok alkalmazásával az explicit típuskonverziók is szükségtelenné válnak:

```
class Valami<N> {
   ...
   N n = ...;
   Iterator i = ((List) n).iterator ();  // futásidejű típusellenőrzés
   ...
}
```

```
class Valami<N extends List> {        // fordításidejű típusellenőrzés
   ...
   N n = ...;
   Iterator i = n.iterator ();        // nincs explicit típuskonverzió
   ...
}
```

Paraméterezett típusdeklarációk IV.

- A korlátoknak páronként különbözőnek kell lenniük.
- A korlátként megadott típusok sorrendje nem számít, de ha van osztály típusú korlát, annak az első helyen kell állnia.
- Ha egy típusváltozónak több korlátja van, az esetlegesen ütköző (azonos szignatúrájú, eltérő visszatérési értékű) metódusokra való hivatkozás fordítási hibát okoz.

```
class Test<A extends Test1&Test2> {
    ...
    A a = ...;
    a.method ();                                    // fordítási hiba
    ...
}

interface Test1 { int method (); }
interface Test2 ( void method (); }
```

Paraméterezett típusdeklarációk V.

- A típusváltozók az azt deklaráló teljes osztályban illetve interfészben láthatók, **kivéve a statikus metódusokat és a statikus inicializátorokat**.
- A fentiekből következik, hogy a típusváltozók deklarációjában lehet körbehivatkozás is:

```
interface Convertible<A> {
   ...
   A convertTo ();
}

class Converter<A extends Convertible<B>,B extends Convertible<A>> {
   ...
}
```

Elnevezési konvenciók

- E – elem
- K – kulcs
- N – szám
- T – „típus”
- V – érték
- S, U, V, … - további típusok

Határozatlan típusparaméterek I.

- Ha egy típusparaméter konkrét értékére nincs szükség, használhatunk **határozatlan típusparaméter**eket:

```
public void printList (List<?> l) {
   for (Iterator<?> i = l.iterator ();i.hasNext ();)
      System.out.println (i.next ());
}
```

- A fenti metódus segítségével tetszőleges elemtípusú lista elemeit kiírathatjuk.
- A **?** joker mellett megadhatunk felső, illetve alsó korlátot is az **extends** és a **super** kulcsszóval.

Határozatlan típusparaméterek II.

- Példák korlátok használatára:

```
class MyList<E> {
    ...
    // A lista elemeit hozzáadja a MyList listához.
    public void addAll (List<? extends E> list) { ... }
    ...
}
```

```
// A lista minden elemének értékül adja value-t.
public <T> void assign (T value,List<? super T> list) {
    ...
}
```

Típustörlés

- A J2SE 1.5-ben alkalmazott megvalósítás az ún. **típustörlés** (type erasure) elvén működik.
- A típustörlés azt jelenti, hogy generikus típusokat csak a fordító tart nyilván, futásidőben nem jelennek meg a generikus típusok.
- Egy paraméterezett típusból a típustörlés eredményeképpen kapott típus (erasure, **törölt típus**) az alábbi lesz:
  - Ha egy típusváltozónak **nincs korlátja**, a típusváltozó értéke Object lesz.
  - Ha egy típusváltozónak **van osztály típusú korlátja**, a típusváltozó értéke az adott osztály típus lesz.
  - Ha egy típusváltozónak **csak interfész típusú korlátai** vannak, a típusváltozó értéke az (lexikografikusan) első interfész típus lesz.

Nyers típusok I.

- Ha egy generikus típust a típusváltozók értékeinek megadása nélkül használunk, az ún. **nyers típus**t (raw type) kapjuk, ez lényegében megegyezik a típustörléssel kapott típussal.
- A nyers típus biztosítja az adott osztály generikus és nem generikus verziója közötti átjárhatóságot (korábbi verziókkal való kompatibilitás).
- A nyers típusok teszik lehetővé például azt, hogy a korábbi Java verziók alatt fordított programok fussanak az új, generikus könyvtárakat tartalmazó futtató környezettek alatt.

Nyers típusok II.

- A nyers típusok használatánál fontos tudni, mi lesz az adott paraméterezett típus típustörlésének eredménye:

```
class Generic<A,B extends List> {
   ...
   public A m1 () { ... }   // a visszatérési érték futásidőben Object
   public void m2 (A a) { ... }       // "a" típusa futásidőben Object
   public void m3 (B b) { ... }         // "b" típusa futásidőben List
   ...
}
```

```
Generic x = new Generic<Integer,LinkedList> ();          // nyers típus
x.m1 ();                                                 // OK, Object érték
x.m2 ("alma");                  // OK (Object-ből származik), de warning
x.m3 (new LinkedList ());     // OK (implementálja List-et), de warning
x.m3 (new Integer (1));                                  // fordítási hiba
```

Generikus metódusok I.

- A metódusok az osztályokhoz és interfészekhez hasonlóan elláthatók típusparaméterekkel.
- A típusparamétereket a metódus visszatérési értéke előtt kell feltüntetni.

```
public <E> void csere (E[] a,int i,int j) {
   E t = a [i];
   a [i] = a [j];
   a [j] = t;
}
```

- A metódus hívásakor a típusparamétereket nem kell megadni, a fordító egy típuslevezetési mechanizmus (type inference) segítségével automatikusan meghatározza a típusokat a paraméterek típusai alapján.

Generikus metódusok II.

- A típuslevezetéssel nem meghatározható típusparaméterek értéke Object lesz.

```
<T> List<T> createList (T element) {
   List<T> l = new LinkedList<T> ();
   l.add (element);
   return l;
}
```

```
createList ("listaelem");                    // List<String>
createList (null);                           // List<Object>
```

Leszármaztatás I.

- A generikus típusok létezése szükségessé teszi a felüldefiniált (vagy elrejtett) metódusok visszatérési értékére vonatkozó szabály módosítását.
- Két metódus szignatúrája akkor egyezik meg, ha nevük, paramétereik száma és típusa, típusparamétereik száma, illetve azok korlátai megegyeznek.
- A korábbi JLS szerint a felüldefiniált (vagy elrejtett) metódusok visszatérési értéke változatlan kell maradjon.
- Az új specifikáció szerint **egy metódus visszatérési értéke az összes, általa felüldefiniált (vagy elrejtett) metódus visszatérési értékének leszármazottja kell legyen**.

Leszármaztatás II.

- Néhány példa legális és illegális felüldefiniálásra:

```
class Test1 {
   int m1 () { ... }
   Number m2 () { ... }
}

class Test2 extends Test1 {
   String m1 () { ... }                    // fordítási hiba
   Integer m2 () { ... }                                  // OK
   String m2 () { ... }                  // fordítási hiba
}
```

Leszármaztatás III.

- Azt a speciális esetet, amikor a visszatérési érték típusa az ősben és a leszármazottakban is megegyezik a deklaráló osztály típusával, **kovariáns visszatérési érték**nek nevezzük (covariant return type).

```
class Test1 {
   public Test1 get () { ... }
}

class Test2 extends Test1 {
   public Test2 get () { ... }
}
```

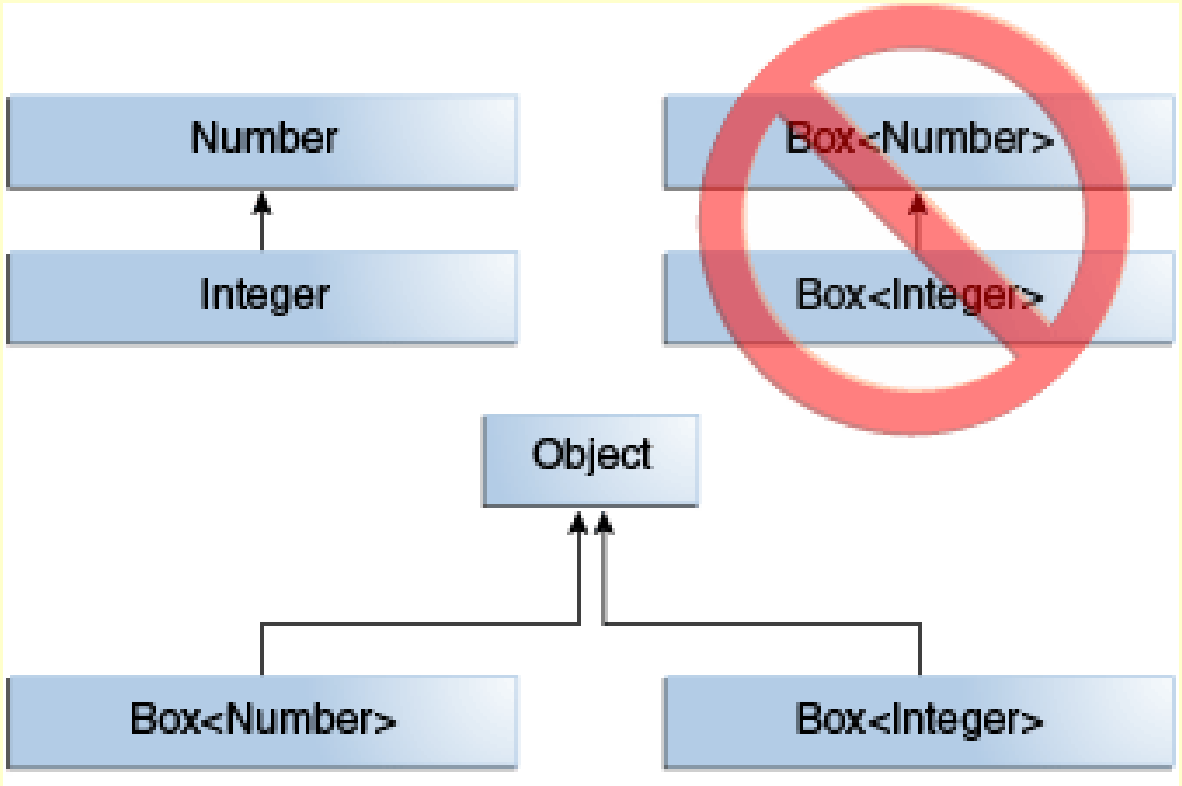
Leszármaztatás IV.

- Generikus típusokat is leszármaztathatunk egymásból:

```
class Test1<A> { ... }

class Test2 extends Test1 { ... }      // OK, nyers típus (A = Object)
class Test2 extends Test1<Integer> { ... }                          // OK
class Test2<A> extends Test1<A> { ... }   // OK, paraméterezzük az őst
```

```
class Test1<A> {
   A m () { ... }
}

class Test2<A> extends Test1<Integer> {
   A m () { ... }   // fordítási hiba, A nem az Integer leszármazottja
}

class Test3<A extends Integer> extends Test1<Integer> {
   A m () { ... }                                                   // OK
}
```

Leszármaztatás V.



Forrás: oracle.com

Kivételkezelés I.

- Mivel futásidőben a generikus típusinformáció nem jelenik meg, a catch blokkok fejében nem szerepelhetnek típusváltozók, illetve paraméterezett típusok.

```
class MyException<E> extends Exception { ... }        // fordítási hiba
```

```
public void m () {
   ...
   try {
      ...
   } catch (MyException<Integer> e) { ... }           // fordítási hiba
   ...
}
```

Kivételkezelés II.

- A throws deklarációkban szerepelhetnek típusváltozók.

```
class Test<T extends Throwable> {

   public void m () throws T { ... }
}
```

```
public <T extends Throwable> void m (Test<T> t) throws T { ... }
```

Tömbök és paraméterezett típusok

- Paraméterezett elemtípusú tömbök nem hozhatók létre, bár deklarálhatók:

```
List<String>[] sl;
sl = new LinkedList<String> [10];                    // fordítási hiba
```

- Ennek oka az, hogy a típustörlés miatt a tömb elemtípusa futásidőben nem lenne ismert, és így nem lehetne védekezni az elemtípusnak nem megfelelő elemek betétele ellen (ArrayStoreException).
- Nyers típusú tömb viszont lehetséges, bár veszélyes:

```
List<String>[] sl = new LinkedList [10];                // nyers típus
sl [0] = new LinkedList<Number> ();                  // fordítási hiba
List[] ol = sl;
ol [0] = new LinkedList<Number> ();
...
String s = sl [0].get (0);                        // ClassCastException
```

Megkötések

- Nem használhatóak primitív típusok típus paraméterként
- Nem lehet a típusparamétert példányosítani (típustörlés), megoldás lehet a reflection használata (lásd később)
- Statikus mezőnek nem lehet generikus típusa
- Típusparaméterre nem hívható az `instanceof` operátor
- Nem lehet paraméterezett elemtípusú tömböt létrehozni
- Nem lehet paraméterezett kivételtípusokat készíteni, eldobni vagy elkapni
- Nem lehet túlterhelni egy metódust ugyanolyan nyers típusú bemenettel