

Programozási nyelvek Java

Kitlei Róbert

Programozási nyelvek és Fordítóprogramok tanszék
ELTE Informatikai Kar

Bevezetés

Példa: rendezett pár

- a rendezett pár adatszerkezet sokféleképpen nézhet ki

```
class Pair { String s; char c; }  
class Pair { double[] ds; boolean b; }
```

- probléma: a különböző fajta pár típusoknak nem lehet azonos a neve

```
class PairSC { String s; char c; }  
class PairDB { double[] ds; boolean b; }
```

- külön nevekkel már használható
 - ◇ kényelmetlen: kézzel kell többfajta osztályt elkészítenünk
 - ◇ a kód későbbi változásait is kézzel kell átvezetni mindenhol

```
PairSC pSC = new PairSC();  
PairDB pDB = new PairDB();
```

Sablon

- a sablon célja hasonló a függvényéhez: csökkenti a kód redundanciáját
 - ◊ metódus: érték paraméterez (metódus)törzsset
 - ◊ **sablon** típus (generic type, template type): típus paraméterez osztályt
- sablonnal csak egy osztályt kell készítenünk

```
class Pair<T1, T2> { T1 elem1; T2 elem2; }
    // ----- a sablon típusparamétereit
```

- konvenció: a típusparaméter neve sokszor rövid, egy-két betűs
 - ◊ T (type): ha általában típusról beszélünk
 - ◊ E (element): egy gyűjteményben elemként használt típus
 - ◊ K és V (key, value): kulcs és érték típusa
- sablon **példányosítása** (instantiation): konkrét paraméterekkel ellátás
 - ◊ a sablonpéldányok használhatóak bárhol, ahol típusokat várunk

```
Pair<String, Character> pSC = new Pair<String, Character>();
Pair<double[], Boolean> pDB = new Pair<>();
// ----- kötelező rész
// Java 7-től megengedett rövidítés ----
```

Sablonparaméterek használata

- a sablonparaméterek megjelenhetnek a műveletekben formális paraméter, lokális változó és a visszatérési érték típusában is

```
class Pair<T1, T2>
{
    T1 elem1;
    T2 elem2;

    T1 getElem1() { return elem1; } // mj. az enkapszuláció
    T2 getElem2() { return elem2; } // itt sérül

    void setElem1(T1 elem1) { this.elem1 = elem1; } // itt is
    void setElem2(T2 elem2) {
        T2 tmp = elem2;
        this.elem2 = tmp; // és itt is
    }
}
```

Sablon példányosítása

- **reification**: a sablonpéldányosítás során milyen implementáció készül?
 - ◇ sok nyelvben minden példányosítás külön, új típust készít
 - ▶ ha a típusparaméterek ugyanolyanok, mint egy másik példányosításnál, akkor a fordítóprogram használhatja az ott készített típusokat is
 - ▶ ilyen nyelvekben a `Pair<String, Character>` a `PairSC` típushoz, a `Pair<double[], Boolean>` a `PairDB` típushoz hasonló típust hoz létre
 - ▶ Javában is vannak rá kísérletek, pl. **jeneral**

Sablon példányosítása

- Javában csak egy implementáció készül
 - ◇ **típus törlés** (type erasure): a típusparaméterek minden előfordulása helyére `Object` kerül a fordítás során
 - ▶ a futtató rendszer nem is tud a típusparaméterekről
 - ▶ törlés előtt a fordító megvizsgálja, hogy típushelyesen használjuk-e a sablonpéldányokat
 - ▶ a típusparaméterek nem lehetnek primitív típusok, mert azok nem helyettesíthetők `Object`-tel
 - a Java sablonok futási időben kevésbé hatékonyak
 - aktuális típusparaméternek bármilyen referenciatípus választható
 - ▶ a típusparaméterekre előírt megkötések alapján a fordítóprogram az `Object`-nél specifikusabb helyettesítőt is találhat

Sablon példányosítása

- a Pair osztály, a minden sablonpéldány számára közös implementáció

```
// Javában ez az ábrázolása minden példányosított Pair-nek
class Pair
{
    Object elem1;
    Object elem2;

    Object getElem1() { return elem1; } // enkapszuláció
    Object getElem2() { return elem2; } // továbbra sincsen

    void setElem1(Object elem1) { this.elem1 = elem1; }
    void setElem2(Object elem2) {
        Object tmp = elem2;
        this.elem2 = tmp;
    }
}
```

Sablon példányosítása

- hasonló megközelítéssel, de típusparaméterek nélkül folyton típuskényszerítéseket kellene alkalmaznunk

```
Pair p = new Pair();  
p.setElem1("abc");  
p.setElem2(new double[]{1, 2, 3});  
String s = (String)p.getElem1();  
double[] ds = (double[])p.getElem2();
```

- ha esetleg elvétjük, futási idejű kivétel váltódik ki
- sablonok használatával...
 - ◇ már fordítási időben hibajelzést kapunk
 - ◇ a típuskényszerítések megmaradnak
 - ▶ a fordítóprogram generálja őket, nem kell kézzel kiírni
 - ◇ ha lefordul a kód, futási időben garantáltan nem kapunk típuskényszerítés miatt kivételt

Sablonhasználat paraméterek nélkül

- **raw type**: sablon típus (kompatibilitási okokból) használható a paraméterei nélkül is
 - ◇ mindig érdemes kiírni a típusparamétereket
 - ◇ a nyelv 2004 óta tartalmazza a sablonokat, csak az annál korábbi verzióknál okozna problémát

```
// működik, de ellenjavallott
```

```
Map nameToBook = new HashMap();
```

```
// szintén működik, és szintén ellenjavallott
```

```
Map nameToBook = new HashMap<String, Book>();
```

```
// a hosszú alak, a Java 5 és a Java 6 csak ezt támogatja
```

```
Map<String, Book> nameToBook = new HashMap<String, Book>();
```

```
// így célszerű
```

```
Map<String, Book> nameToBook = new HashMap<>();
```

Altípusosság

- `List<Integer>` nem altípusa `List<Object>`-nek
 - ◊ ennek ellentmondani látszik, hogy `Integer` altípusa `Object`-nek

```
Object ertek = new Integer(4); // ez persze helyes
```

```
List<Integer> ints = new ArrayList<Integer>(); // (1)
List<Object>  objs = ints; // fordítási idejű hiba
                        // az alábbiaktól óv meg
```

```
objs.add(ertek); // "ertek" dinamikus típusa Integer,
                // ez nem okozna problémát
```

```
Object obj = new Object(); // (2)
objs.add(obj); // objs-ba kerülhetnek Object-ek
Integer ki = ints.get(0); // ints-ből Integer-ek kérhetőek
System.out.println(ki + 1); // (2) Object, nem végezhető számítás
```

Altípusosság: tömbök vs sablonok

- a tömbök a sablonoknál sokkal korábban kerültek a nyelvbe, ezért több szempontból kilógnak a többi adatszerkezet közül
 - ◇ a tömbök *futási időben is tudják*, milyen típusú elemeket tárolnak
 - ▶ a sablonok a típustörlés miatt nem
 - ◇ a tömbök primitív típusú elemeket is tudnak tárolni
 - ◇ a tömbök hossza példányosításkor rögzített, a legtöbb adattípusé nem

```
Integer[] ints = new Integer[5]; // (1)
Object[]  obj = ints;           // szabályos (de nem célszerű)
obj[0] = new Integer(4);       // OK: Integer bekerülhet (1)-be
obj[1] = new Object();         // futási idejű ArrayStoreException
```

- a primitív/referenciatípusok viszont tömbökben sem keveredhetnek

```
int[]     ints = new int[5];
Object[]  obj = ints;           // fordítási idejű hiba
```

Tömbök vs sablonok adatszerkezetekben

- a legtöbb adatszerkezet betehető más adatszerkezetekbe, pl. List Map-be, és problémamentesen használhatóak együtt

```
List<Integer> list1 = new ArrayList<>(Arrays.asList(1, 2, 3));
List<Integer> list2 = new ArrayList<>(Arrays.asList(1, 2, 3));
```

```
Map<List<Integer>, String> tomb2 = new HashMap<>();
tomb2.put(list1, "123");
System.out.println(tomb2.get(list2)); // "123", ahogy vártuk
```

- a tömbökre nincsen rendszeresen megírva az equals és a hashCode művelet
 - ◊ az Object-től örökölt implementációkat használják
 - ▶ nem a tartalmakkal dolgoznak, hanem az objektumok referenciáival

```
Map<int[], String> tomb1 = new HashMap<>();
tomb1.put(new int[]{1,2,3}, "123");
System.out.println(tomb1.get(new int[]{1,2,3})); // null !
```

- tanulság: adatszerkezetekbe ne tegyünk tömböket

Példányok készítése

- Javában nincsen se List, se Set, se Map literál
- általános megoldás: példányosítsunk üreset, majd elemenként töltsük fel

```
Set<String> elems = new HashSet<>();  
elems.add("a");  
elems.add("b");  
elems.add("c");
```

- a listákhoz az előző fólián szereplő trükköt lehet használni
 - ◇ (általában elfogadható) többletköltség: +1 tömb és +1 lista is elkészül

```
new ArrayList<>(Arrays.asList(/* az elemek felsorolása */))
```

- ha csak felhasználni akarjuk a listát, elég a belső rész
 - ◇ az asList egy névtelen ArrayList-leszármazottal tér vissza, amelyen nem támogatott minden művelet

```
List<Integer> elems = Arrays.asList(1, 2, 3);  
for (Integer elem : elems) System.out.println(elem); // OK  
elems.add(4); // UnsupportedOperationException
```

Megkötések a típusparaméterekre

- **bounded type parameter** (korlátozott típusparaméter): a típusparaméterekre megkötéseket is tehetünk
- a legegyszerűbb megkötés: a típusparaméternek egy adott típus leszármazottjának kell lennie
 - ◇ a saját műveleteink ennek a típusnak a műveleit használhatják

```
class Foo<T extends PrintWriter> {
    T t;
    public Foo(T t) { this.t = t; }
    void doPrint(String s){t.println(s + "Foo"); t.flush();}
    // itt használjuk -----
```

```
class PW2 extends PrintWriter {
    public PW2(OutputStream os) throws Exception {super(os);}
    public void println(String s) { super.println(s + s); } }
```

```
OutputStream os = new FileOutputStream("x.txt");
Foo<PW2> myFoo = new Foo<>(new PW2(os));
myFoo.doPrint("abc"); // x.txt tartalma: abcFooabcFoo
```

Megkötések a típusparaméterekre

```
// T leszármazottja Exception-nek, valamint  
// megvalósítja a Cloneable és a Collection<Integer> interfészt  
class Foo<T extends Exception  
    & Cloneable  
    & Collection<Integer>> { ... }
```

Megkötések a típusparaméterekre

// a megadott típusnak lehet típusparamétere

```
class Foo<T, Cmp extends Comparator<T>> {
    // akár másik típuspar. is lehet ---
    T t; Cmp cmp;

    public Foo(T t, Cmp cmp) { this.t = t; this.cmp = cmp; }

    public int compareToCenter(T other) {
        return cmp.compare(other, t);
    }
}
```

```
Foo<String, Comparator<String>> foo =
    new Foo<>("abcde", (s1, s2) -> s1.length() - s2.length());
System.out.println(foo.compareToCenter("abc"));           // -2
System.out.println(foo.compareToCenter("equal"));         // 0
System.out.println(foo.compareToCenter("zzzzzz"));       // 1
```


Megkötések a típusparaméterekre

// ... sőt, a típusparaméter lehet maga T is!

```
class Foo<T extends Comparable<T>> {
    T t;
    public Foo(T t) { this.t = t; }
    public int cmpTo(T other) { return t.compareTo(other); }
}
```

```
class MyString implements Comparable<MyString> {
    String s;
    public MyString(String s) { this.s = s; }
    public int compareTo(MyString other) {
        return s.compareTo(other.s); }
}
```

```
Foo<MyString> foo = new Foo<>(new MyString("abcde"));
System.out.println(foo.cmpTo(new MyString("abc"))); // poz.
System.out.println(foo.cmpTo(new MyString("equal"))); // 0
System.out.println(foo.cmpTo(new MyString("zzzzzz"))); // neg.
```

Megkötések a típusparaméterekre

// akár ez is megengedett

```
class Foo<T1, T2 extends T1> { ... }
```

// fordítási hiba,

// a paraméter típusok nem tartalmazhatnak kört

```
class Foo<T1 extends T2, T2 extends T1> { ... }
```

Megkötések a típusparaméterekre

- korábban volt szó róla: az alábbi helytelen

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

- természetesen ez paraméterátadás során is helytelen

```
void f(List<Object> objs) {
    for (Object obj : objs)     System.out.println(obj);
}
```

```
f(new ArrayList<String>());    // ez is fordítási hiba
f(new LinkedList<String>());  // ... és ez is
```

- márpedig így nem tudunk tetszőleges listát kiírni, amire szükségünk lenne

Megkötések a típusparaméterekre

- megoldás: **wildcard** (dzsóker, helyettesítő) típusparaméter
 - ◇ ? egy tetszőleges típust jelöl
 - ▶ List<?> bázistípusa ArrayList<String>-nek ?←String választással

```
void f(List<?> objs) {  
    for (Object obj : objs)      System.out.println(obj);  
}
```

```
f(new ArrayList<String>());    // így már nem fordítási hiba  
f(new LinkedList<String>());   // ... és ez sem
```

- a ?-t legtöbbször adatszerkezetek felhasználásánál alkalmazzuk, de feltölteni nem lehet vele őket

```
List<?> objs = new ArrayList<String>(); // érvényes  
objs.add("abc");                       // fordítási hiba  
objs.add(new Object());                 // ez is érvénytelen
```

Megkötések a típusparaméterekre

- wildcard paraméterekre további megkötéseket is lehet tenni

```
void mySort(List<? extends Comparable> objs) {  
    java.util.Collections.sort(objs);  
}
```

// ez helyes

```
mySort(new ArrayList<String>());
```

// ez nem, ha a MyClass

// nem valósítja meg a Comparable interfészt

```
mySort(new LinkedList<MyClass>());
```

Metódusok típusparaméterei

- az osztályok mellett metódusok is kaphatnak típusparamétereket

```
class C {
    static <T> boolean noSameObjs(List<T> elems) {
        // --- a típusparamétert a fejléc elé kell írni
        //      (példányszintű metódusok esetén is)
        return elems.size() == new HashSet<T>(elems).size();
    }

    public static void main(String[] args) {
        List<String> texts = new ArrayList<>();
        texts.add("abc");
        System.out.println(C.<String>noSameObjs(texts));
        // a teljes alak híváskor -----
        // ha a rendszer ki tudja következtetni, lehet rövidíteni
        System.out.println(noSameObjs(texts)); // true
        texts.add("abc");
        System.out.println(noSameObjs(texts)); } // false
    }
}
```