Programozási nyelvek Java

11.gyakorlat

Operációsrendszertől függő tulajdonságok

PATH elválasztó

- Unix – ":"
- Windows – ";"
- final String PATH_SEPARATOR = File.pathSeparator;
- Ugyanaz, csak karakterkent final char PATH_CHAR = File.pathSeparatorChar;

Név szeparátor

- Unix – "/"
- Windows – "\"
- final String SEPARATOR = File.separator;
- Ugyanaz, csak karakterkent final char SEPARATOR_CHAR = File.separatorChar;

Sorvége karakter

- Unix – "\n"
- Windows – "\r\n”
- final String EOL = System.getProperty("line.separator");

Generic (Sablon)

Osztálysablon készítésére alkalmas. Általánosan megírt sablonba, használatkor konkrét típusokat helyettesítünk be.

- Típussal paraméterezhető
- Type erasure: csak fordítási időben ismert a típusinformáció, utána automatikusan törli a fordító, bájtkódból nem szerezhető vissza
- Nem kötelező velük foglalkozni (@SupressWarnings("rawtypes", "unchecked")), de rendkívül hasznosak, fordítási időben tudunk potenciális hibalehetőségeket kiszúrni - persze ez is a programozón múlik
- Kényelmes, típusbiztos
- Collection-öknél aktívan használjuk őket: LinkedList<String> s = new LinkedList<String>();
- Részletesen: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Példa

```
//1.5 előtt
LinkedList l = new LinkedList();
l.add(new Integer(1));
l.add(new Integer(2));

for(int i=0; i<l.size(); ++i)
{
    Integer curr = (Integer)l.get(i);
    System.out.println(curr);
}

//Azóta
LinkedList<Integer> l = new LinkedList<Integer>();
l.add(1);
l.add(2);

for(int i=0; n=l.size(); i<n; ++i)
{
    Integer curr = l.get(i);
    System.out.println(curr);
}
```

Autoboxing-unboxing

- Autoboxing: automatikusan becsomagolja //pl: Integer i = 5;
- Unboxing: kicsomagolás a csomagoló osztályból //pl: int i = new Integer(5);
- Csak objektum referenciákat tárolhatnak, ezért primitív típusok helyett csomagoló (wrapper) osztályokat (Integer, Character, Double, etc.) kell használnunk - azonban ezek ilyen esetekben automatikusan konvertálódnak (ld. fenti példa)
- Amire figyelni kell: teljesítmény, == operátor, null unbox-olása NullPointerException-nel jár

```
Integer i = new Integer(5);
int j = i;
j = new Integer(5);
j = i.intValue();
i = 5;
```

Altípusosság

```
List<String> l1 = new ArrayList<String>();
List<Object> l2 = l1; //hiba
```

Nem konvertálhatók, mert akkor lehetne ilyet csinálni:

```
l2.add(new Object());
l1.get(0); //futási időben hiba, Object -> String cast-olas miatt
```

Egyszerűbb példák

A java.util csomagban találhatóak a következő példák:

```
public interface List<T>
{
    void add(T e);
    Iterator<T> iterator();
}

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

A T formális típusparaméter, amely aktuális értéket a kiértékelésnél vesz fel (pl. Integer, etc.).

Wildcardok

Általános megoldást szeretnénk, amely minden collection-t elfogad, függetlenül az azokban tárolt elemektől (pl. ki szeretnénk őket írni), vagy nem tudjuk azok konkrét típusát (pl. legacy code). Collection<Object> nem őse (ld. előző bekezdés).

Ha nem használunk generic-ket, akkor is megoldható, viszont warning-ot generál:

```
public void print(Collection c)
{
    for(Object o : c)
    { System.out.println(o); }
}
```

A megoldás a **wildcard** használata: Collection<?> minden kollekcióra ráillik.
Ilyenkor Object-ként hivatkozhatunk az elemekre:

```
public void print(Collection<?> c)
{
    for(Object o : c)
    { System.out.println(o); }
}
```

Vigyázat!!! A ? ≠ Object! Csak egy ismeretlen típust jelent.
Így a következő kódrészlet is fordítási hibához vezet:

```
List<?> c = ...;
l.add(new Object());  //fordítási hiba
```

Nem tudjuk, hogy mi van benne, lekérdezni viszont lehet (mert tudjuk, hogy minden objektum az Object leszármazottja).

Bounded wildcard

Amikor tudjuk, hogy adott helyen csak adott osztály leszármazottai szerepelhetnek, akkor használhatjuk őket.

Első megközelítés (ROSSZ!!!):

```
abstract class Super {}
class Sub1 extends Super {}
class Sub2 extends Super {}
...
void func(List<Super> l) {...}  //Rossz!!!
```

Azért rossz, mert a func() csak List<Super>-rel hívható meg, List<Sub1>, List<Sub2> nem lehet paramétere (nem altípus).

Megoldás a **bounded wildcard**:

```
abstract class Super {}
class Sub1 extends Super {}
class Sub2 extends Super {}
...
void func(List<? extends Super> l) {...}
```

Belepakolni ugyanúgy nem tudunk, mint a ? esetén, azaz erre fordítási hibát kapunk:

```
void func(List<? extends Super> l)
{
    l.add(new Sub1()); // fordítási hiba
}
```

Felfelé is megköthető a wildcard a <? super T> jelöléssel.

Példa

Generikus osztály

```
public class Pair<T1, T2>
{
    private final T1 first;
    private final T2 second;

    public Pair(final T1 first, final T2 second)
    {
        super();
        this.first = first;
        this.second = second;
    }

    public T1 getFirst()
    { return first; }

    public T2 getSecond()
    { return second; }

    @Override
    public String toString()
    { return "(" + first + ", " + second + ")"; }

    @Override
    public boolean equals(Object o)
    {
        if (null == o) return false;
        if (this == o) return true;
        if (this.getClass() != o.getClass()) return false;

        Pair p = (Pair)o;
        return this.first.equals(p.first) && this.second.equals(p.second);
    }

    @Override
    public int hashCode()
    {
        int hash = 7;
        hash = 31 * hash + (null != this.first ? this.first.hashCode() : 0);
        hash = 31 * hash + (null != this.second ? this.second.hashCode() : 0);
        return hash;
    }
}
```

Generikus függvény

```
public class GenericFunction
{
    public static <T extends Comparable<T>> T min(T item1, T item2)
    {
        T item = null;
        switch(item1.compareTo(item2))
        {
            case -1: item = item1; break;
            case 0: item = item1; break;
            case 1: item = item2; break;
        }

        return item;
    }

    public static void main(String[] args)
    {
        System.out.println(min(5,7));
        System.out.println(min(9,7));
        System.out.println(min(6,6));
    }
}
```

Feladatok

1. Készíts egy generikus **reverse** függvényt, amely egy tetszőleges lista elemeit képes megfordítani.

2. Készíts egy generikus **addAll** függvényt, amely egy tetszőleges lista adatszerkezetbe beleteszi a második paraméterként megadott lista összes elemét!

3. Készíts egy olyan generikus **min** függvényt, amely egy tetszőleges kollekcióban megkeresi és visszaadja a minimális elemet! (Kell a Comparable interfész is!!!).

4. Készíts egy saját, generikus **Stack** implementációt! Az implementáció módját tetszőlegesen megválaszthatod, a reprezentációt is, de illeszkedjen az IStack interfészre! Az implementációk a generic.stack csomagban legyenek!

   Teszteljétek a Main fájl alapján!

5. Készítsd el az előző **Stack** adatszerkezetet úgy, hogy az elemeket Node típusban tárolod. A node egy olyan adatszerkezet, ami tartalmaz egy elemet és 2 referenciát: az előző és a következő node-ra. Ezt megtalálod a honapon.

6. Készítsd el a MyList generikus osztályodat úgy, hogy származtatod a LinkedList generikus osztályból. Az osztályodat egészítsd ki két függvénnyel:
   public T first(Predicate<T> pred) és a **public LinkedList<T> filter(Predicate<T> pred)**
   A first függvény visszaadja az első olyan elemet, amire a megadott predikátum test függénye igazzal tér vissza, míg a filter függvény egy listába rakja azokat az elemeket, amire igaz feltétel, majd visszatér vele.

   Készíts egy tesztfájlt, amiben teszteled a MyList osztályodat Integer típusra, a predikátumbeli test függvényed pedig a páros számokra térjen vissza igazzal.

7. Készítsünk egy egyszerű, általános **bináris keresőfa** implementációt! A fához lehessen elemet hozzáadni (add()), kiírni, valamint a minimum, maximum elemet megkeresni (min(), max()). A típusparaméterének összehasonlíthatónak kell lennie (< T extends Comparable<T> >).