A programozás alapjai 3.

Java genericitás és kollekciók

Goldschmidt Balázs
balage@iit.bme.hu

1

Genericitás

2

Genericitás

- Cél: kód újrahasznosítása különböző típusokkal
  - egységes megoldás típushelyes működéssel
- C megoldása
  - `void* malloc(size_t s)`
    - a kasztolás veszélyes
    - maradhatnak hibák (nincs típusellenőrzés)
  - `#define max(a,b) ((a)>(b)?(a):(b))`
    - mellékhatások esetén hibás eredmény

Genericitás

- Cél: kód újrahasznosítása különböző típusokkal
  - egységes megoldás típushelyes működéssel
- C++ megoldás
  - ```
    template <class T> T max(T a, T b) {
      return (a>b)?a:b;
    }
    ```
    - az elvárások a `T` típussal szemben nem explicitek
      - dokumentáció fontos
    - hibák fordítási időben
    ```

Genericitás

- Cél: kód újrahasznosítása különböző típusokkal
  - egységes megoldás típushelyes működéssel
- **`Java`** eredeti megoldása (pre 5.0)
  - minden osztály az *Object* leszármazottja
    - a **`C void*`** megoldása OO köntösben
    - kód tele lesz kasztolással
    - hibák egy része csak futási időben derül ki
    - típusellenőrzés: *instanceOf* operatorral

```
if (s instanceof String) ...
```


5

Genericitás példa Object-tel

```
public class Store {
  Object[] os; int size;
  public Store(int i) {
    os = new Object[i];
    size = 0;
  }
  public void put(Object o) {
    os[size] = o; size++;
  }
  public Object get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

```
Store s1 = new Store(10);

s1.put("hello ");
s1.put("world");
s1.put("!");

for (int i = 0;
     i < s1.size();
     i++) {

  String l = (String)s1.get(i);
  l = "   " + l;
  System.out.print(l);

}
```

Kasztolás a kliensben ☹

6

Genericitás példa Object-tel

```
public class Store {
  Object[] os; int size;
  public Store(int i) {
    os = new Object[i];
    size = 0;
  }
  public void put(Object o) {
    os[size] = o; size++;
  }
  public Object get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

```
Store s2 = new Store(10);

s2.put(13.1);
s2.put(14.2);
s2.put(15.3);

for (int i = 0;
     i < s2.size();
     i++) {

  Double d = (Double)s2.get(i);
  d *= 3.0;
  System.out.print(d);

}
```

Kasztolás a kliensben ☹

Java generikus osztályok

- Java 5 óta
- Hasonló, mint a C++ sablonok
  - *De: nem generálunk újabb osztályt minden egyedi sablon-paraméterhez*
- Sablon-paraméter interfésze kötött
  - a fejlesztő a fejlécből látja
- Hibák fordítási időben előjönnek
- Kasztolás-mentes forráskód
- A sablonparaméter nem lehet primitív típus
  - csomagoló-osztályok!

Genericitás példa sablonnal

```
public class Store<T> {
  T[] os; int size;
  public Store(int i) {
    os = (T[])(new Object[i]);
    size = 0;
  }
  public void put(T o) {
    os[size] = o; size++;
  }
  public T get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

Kis trükk tömbökhöz

```
Store<String> s1 =
       new Store<String>(10);

s1.put("hello ");
s1.put("world");
s1.put("!");

for (int i = 0;
     i < s1.size();
     i++) {

  String l = s1.get(i);
  l = "   " + l;
  System.out.print(l);

}
```


Genericitás példa sablonnal

```
public class Store<T> {
  T[] os; int size;
  public Store(int i) {
    os = (T[])(new Object[i]);
    size = 0;
  }
  public void put(T o) {
    os[size] = o; size++;
  }
  public T get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

```
Store<Double> s2 =
       new Store<Double>(10);

s2.put(13.1);
s2.put(14.2);
s2.put(15.3);

for (int i = 0;
     i < s2.size();
     i++) {

  Double d = s2.get(i);
  d *= 3.0;
  System.out.print(d);

}
```

Genericitás és öröklés (tömbök)

```
Object[] oa = new String[10];
oa[0] = "Hello";
oa[1] = new Integer(2);
      // RT error: ArrayStoreException
```

- `Object[]` helyettesíthető `String[]`-bel
- Csak a dinamikus típussal kompatibilis objektumok tárolhatók

Genericitás és öröklés

```
// bővítsük a Store<T> osztályt
void printStore(Store<Object> of) {
  for (int i = 0; i < of.size(); i++) {
    System.out.println(of.get(i));
  }
}
```

```
// hívjuk meg az új metódust!
Store<String> s = new Store<String>();
Store<Integer> i= new Store<Integer>();

s.printStore(i); // CT error!
```

Genericitás és öröklés

```
// ugyanez kibontva

Store<String> s = new Store<String>(10);
Store<Object> o = s; // CT error!
o.put(new Integer(13)); // nem kompatibilisek!
```

- **`Store<Object>` nem** a `Store<String>` őse!
- A típusaik inkompatibilisek!

Dzsóker: `?`

```
void printStore(Store<?> of) {
 for (int i = 0; i < of.size(); i++) {
    System.out.println(of.get(i));
  }
} // ez már OK
```

```
Store<String> sf = new Store<String>();
Store<?> of = sf;
of.put(new Integer(13));  // CT error
of.put("Hello");          // CT error
```

- **`<?>`** kifele bármivel kompatibilis
  - a belőle kivett objektumok típusa *Object*
  - belerakni semmit sem lehet

Kötött dzsóker: *leszármazottak*

```
// bővítsük a Store<E> osztályt
void putAll(Store<E> st) {
  for(int i = 0; i < s.size(); i++) {
      put(st.get(i));
}
```

- Mi történik, ha **`st`** típusa **`Store<Q>`**, ahol **`Q`** *leszármazottja* **`E`**-nek?

```
Store<Person> p  = new Store<Person>();
Store<Person> p2 = new Store<Person>();
Store<Student> s = new Store<Student>();
...
p.putAll(p2); // OK
p.putAll(s);  // CT error
```


Kötött dzsóker: *extends*

```
// bővítsük a Store<E> osztályt
void putAll(Store<? extends E> st) {
  for(int i = 0; i < s.size(); i++) {
      put(st.get(i));
}
```

- Ezzel *st* típusa ellenőrizhetővé vált
  - *st* tartalma *E* vagy *E* leszármazottja

```
Store<Person> p  = new Store<Person>();
Store<Person> p2 = new Store<Person>();
Store<Student> s = new Store<Student>();
...
p.putAll(p2); // OK
p.putAll(s);  // OK
```

Kötött dzsóker: *ősosztályok*

```
// bővítsük a Store<E> osztályt
void put2All(Store<E> st) {
  for(int i = 0; i < size(); i++) {
      st.put(get(i));
}
```

- Mi történik, ha **`st`** típusa **`Store<Q>`**, ahol **`E`** *leszármazottja* **`Q`**-nak?

```
Store<Student> s  = new Store<Student>();
Store<Student> s2 = new Store<Student>();
Store<Person>  p  = new Store<Person>();
...
s.put2All(s2); // OK
s.put2All(p);  // CT error
```


Kötött dzsóker: *super*

```
// bővítsük a Store<E> osztályt
void put2All(Store<? super E> st) {
  for(int i = 0; i < size(); i++) {
      st.put(get(i));
}
```

- Ezzel *st* típusa ellenőrizhetővé vált
  - *st* tartalma *E* vagy *E* őse

```
Store<Student> s  = new Store<Student>();
Store<Student> s2 = new Store<Student>();
Store<Person>  p  = new Store<Person>();
...
s.put2All(s2); // OK
s.put2All(p);  // OK
```

Több típusú sablon-paraméterek

- Hogyan jelöljük, hogy a paraméter egyszerre kell **`X`** és **`Y`** típusú legyen?

```
<T extends X & Y>
```

```
public static
<T extends Object & Comparable<? Super T>>
T max(Collection<? extends T> coll)
```


Generikus típusok szabályai

- Primitív típus nem lehet sablonparaméter

```
Store<int> st = new Store<int>(); // CT error
```

- Sablonparaméter nem példányosítható

```
T t = new T(); // CT error
```

- Nem lehet statikus sablonparaméter attribútum

```
static public T test; // CT error
```

```
Store<String>.test = "hello";
Store<Integer>.test = 13;     // CT error
```

Generikus típusok szabályai

- Nem használható az *instanceof*

```
void foo(Store<T> s) {
  if (s instanceof Store<String>) ... // CT error
```

- Generikus típushoz nincs tömbpéldány

```
Store<String>[] l1, l2, l3;
l1 = new Store<String>[10];           // CT error
l2 = (Store<String>[])new Object[10]; // CT error
l3 = (Store<String>[])new Store[10];  // OK
```

  - A kasztolós megoldás elkerülendő

Generikus típusok szabályai

- Nincs felüldefiniálás sablonparaméterrel
  - ugyanarra a típusra kötődhet (pl. *S* = *T* = *String*)

```
class Generic<S,T> {
   public void foo(S s) {...}
   public void foo(T t) {...} // CT error
```

- Kivételkezelés korlátozott
  - Nincs generikus kivételtípus (*catch* felüldefiniálás)
  - Nincs sablonparaméteres *catch*
  - El lehet dobni sablonparamétert

Kollekciók

Kollekciók: a tömb

- beépített típus
- immutábilis méret
  - C/C++ nem ellenőrizte, volt **`realloc`**
  - Java ellenőrzi, nincs **`realloc`**
    - nem tudjuk elkerülni, hogy referenciákat másoljunk
- öröklés és típuskompatibilitás gondot okozhat

```
Object[] oa = new String[10];
oa[3] = new Double(3.14); // RT error
```

- kényelmesen használható (többnyire)

Kollekciók: dinamikus adatstr-k

- Láncolt lista
  - egyszeres vagy duplán láncolt, gyűrű, strázsa, fésűs
- Bináris (vagy többes) fa
  - egyensúly, vörös-fekete, AVL
  - szó-fa
- Hash-tábla
  - asszociatív tárolás (kulcs-érték)
  - hash-fv fontos (a kulcsoknál)

Kollekciók: általános jellemzők

- Közös alapfunkciók
  - beillesztés, keresés, csere, törlés
    - CRUD: create, read, update, delete
  - iterálás
- Eltérő megvalósítások
  - rendezett
  - halmaz/zsák
  - mély/sekély másolás
  - optimalizálás: pl. beillesztés vagy keresés

Kollekciók használata

```
List<Integer> l = new ArrayList<Integer>();

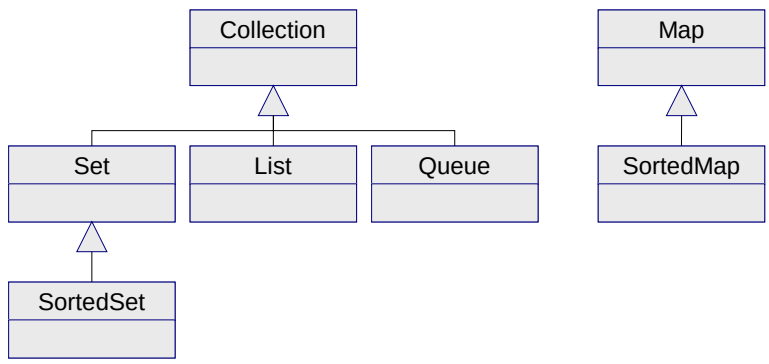
// since J2SE 7
// List<Integer> l = new ArrayList<>();

for (int i = 0; i < args.length; i++) {
      l.add(Integer.parseInt(args[i]));
}

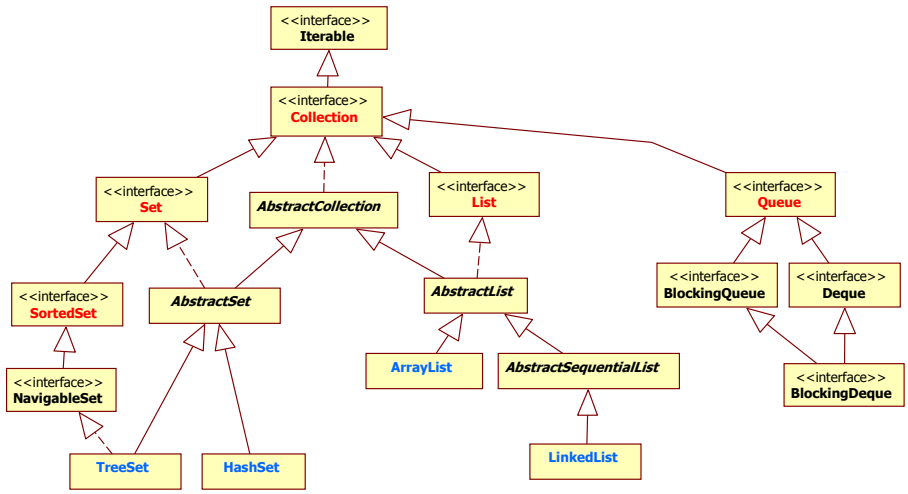
for (int i = 0; i < l.size(); i++) {
      System.out.println(l.get(i)+10);
}
```


Kollekció keretrendszer (részlet)

- Interfészek



Kollekció keretrendszer (részlet2)



Interface `Collection`

- Tipikus kollekció műveletek
- Lehet
  - rendezett vagy sem
  - halmaz vagy zsák
  - néhány metódus opcionális (nem implementált)
- Generikus deklaráció:

```
Collection<E>
```

Interface `Collection` 2

- **`void add(E e)`** *opcionális*
  - objektum beillesztése
- **`void addAll(Collection<? extends E> c)`** *opcionális*
  - `c` kollekció összes elemének beillesztése
  - csak referenciákat másolunk!
- **`boolean remove(E e)`** *opcionális*
  - objektum törlése
- **`boolean removeAll(Col<? ext E> c)`** *opcionális*
  - minden `c`-ben is szereplő objektum törlése
  - csak a referenciákat, az objektumok maguk nem sérülnek!

UnsupportedOperationException

Interface `Collection` 3

- **`boolean contains(E e)`** *opcionális*
  - igaz, ha *e* tárolva van
- **`boolean containsAll(Col<? ext E> c)`** *opcionális*
  - igaz, ha *c* minden eleme tárolva van
- **`int size()`**
  - tárolt objektumok száma
- **`boolean isEmpty()`**
  - igaz, ha üres
- **`void clear()`**
  - minden referencia eltávolítása

Interface `Collection` 4

- **`boolean retainAll(`*`Col`*`<? `*`ext`*` E> c)`** *opcionális*
  - csak a *c*-ben is meglevő objektumok megőrzése
- **`boolean equals(E e)`**
  - tartalmi egyezőség egy másik kollekcióval
  - szimmetrikus művelet (*a.equals(b)* ↔ *b.equals(a)*)
- **`Object[] toArray()`**
  - kollekció tartalma Object tömbként
- **`<T> T[] toArray(T[] ta)`**
  - kollekció tartalma **`ta`**-val megegyező típusú tömbként
- **`Iterator<E> iterator()`**
  - visszaad egy iterátort a kollekcióhoz

Interface `Iterator`

- Minden iterátornak meg kell valósítania
- Elemek visszaadása
  - jelentősen eltér a C++ STL iterátoraitól!
- Deklaráció: **`Iterator<E>`**
- **`boolean hasNext()`**
  - ha van még nem iterált elem
- **`E next()`**
  - visszaadja a következő elemet, ha van
- **`void remove()`**
  - törli az utolsó **`next()`** által visszaadott elemet

Interface `Iterator` 2

- Tipikus használat

```
Collection<Integer> c = ...;
...
Iterator<Integer> i = c.iterator();

while (i.hasNext()) {
       int a = i.next(); // kidobozolás
       if (a < 0) {
              i.remove();
       }
}
```


Interface `Iterator` 3

- Többszörös hozzáférés kezelése
  - iterálás közben a kollekció módosítása hibához vezet
    - ha nem az érintett iterátor végzi a *remove()*-val
  - **`ConcurrentModificationException`** dobódik

```
Collection c = ...;
...
Iterator i1 = c.iterator();
Iterator i2 = c.iterator();
i1.next();
i2.next();
i2.remove();
i1.next();    // itt kapunk kivételt
```

Interface `Set`

- Halmaz: minden objektum csak egyszer
- Rendezés módja ismeretlen
- Iterator nem definiált sorrendben adja vissza
- Nincs a *Collection*-t bővítő metódusa
- Tipikus megvalósítás: **`HashSet`**
  - hatékony működéshez jó hash-függvény kell
  - azonosság ellenőrzése *equals* metódussal

Interface `SortedSet`

- Rendezett halmaz
- Tartalom alapján rendez
  - természetes vagy **`Comparator`**-alapú
    - Természetes: **`Comparable.compareTo(Object obj)`**
      - *Megmondja, melyik kisebb:* this *vagy* obj (-, 0, +)
    - Comparator: **`int compare(Object o1, Object o2)`**
      - *Megmondja, melyik kisebb:* o1 *vagy* o2 (-, 0, +)
- **`Iterator`** a rendezés sorrendjében iterál
- Tipikus megvalósítás: **`TreeSet`**

Interface `List`

- Elemek sorban (lista)
- Ugyanaz az objektum többször is szerepelhet
- Minden elem pozíciója (indexe) ismert
  - elérjük őket az index alapján is
- Kereshető (objektum -> index)
- Van `ListIterator` segédje
  - az *Iterator* leszármazottja extra műveletekkel
- Tipikus megvalósítás: **`ArrayList`**

Interface `List` 2

- A `List<E>` extra metódusai
  - mindegyik az indexeléshez köthető
  - `add(int index, E e)`
  - `E get(int index)`
  - `int indexOf(Object)`
  - `int lastIndexOf(Object)`
  - `E remove(int index)`
  - `boolean remove(Object o)`: csak az elsőt törli
  - `E set(int index, E e)`
  - `List<E> subList(int from, int to)`
  - `...`

Interface `ListIterator`

- Bővített műveletek (Iterátor-hoz képest)
  - index-alapú metódusok
    - **`int nextIndex():`** következő elem indexe
    - **`int previousIndex()`** : előző elem indexe
  - fordított iterálás
    - **`boolean hasPrevious()`** : *hasNext* visszafele
    - **`T previous()`** : *next* visszafele
  - lista módosítása
    - **`set(T t)`** : utoljára visszaadott elem felülírása a listában
    - **`add(T t)`** : a kurzor elé szúrja az elemet

Interface `Map`

- Kulcs-érték párok tárolása
- Minden kulcshoz tartozik egy érték, ami állítható
- Mutábilis kulcsok használata nem javasolt
- Három nézete van
  - kulcsok halmaza
  - értékek kollekciója
  - kulcs-érték párok halmaza
- Tipikus megvalósítás: **`HashMap`**

Interface **`Map`** 2

- Deklaráció: **`Map<K,V>`**
  - **`K`** kulcs típusa
  - **`V`** érték típusa
- Collection interfészben megismerthez hasonló metódusok:
  - **`void clear()`**
  - **`boolean equals()`**
  - **`boolean isEmpty()`**
  - **`int size()`**

Interface **`Map`** 3

- **`V put(K key, V value)`**
  - beilleszti (felülírja) a *key-value* párt
- **`void putAll(Map<? ext K, ? ext V> m)`**
  - beilleszti *m*-ből az összes párt (ha valami már volt, felülírja)
- **`V get(K key)`**
  - visszaadja a *key* kulcshoz tartozó értéket
- **`V remove(K key)`**
  - törli és visszaadja a *key* kulcshoz tartozó értéket

Interface `Map` 4

- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
  - igaz, ha *key* vagy *value* kulcsként illetve értékként benne van
- `Set<K> keySet()`
  - visszaadja a kulcsok halmazát
- `Collection<V> values()`
  - visszaadja az értékek kollekcióját
- `Set<Map.Entry<K,V>> entrySet()`
  - visszaadja a kulcs-érték párok halmazát

Interface `SortedMap`

- Olyan *Map*, ami a kulcsokat rendezve tárolja
  - természetes vagy `Comparator`-alapú rendezés
  - hasonlóan a *SortedSet*-hez
- Nézetek ennek megfelelően (rendezve)
  - `keys(), values(), entrySet()`
  - a nézetek iterátorai rendezve iterálnak
- Tipikus megvalósítás: `TreeMap`

For-each ciklus

- A Java 5 előtt csak *iterator*
  - **`hasNext()`** és **`next()`**
- A Java 5 óta egyszerűű *for* ciklus (tömbökre is)
  - mindenhez, ami megvalósítja az *Iterable* interfészt

```
Collection<Integer> c = ...;
...
for (Integer i : c) {
       System.out.println(i);
}
```

```
static public void main(String[] args) {
  for (String s : args) { System.out.println(s); }
}
```


Kollekció példa

- Írjuk ki egy szövegből a betűgyakoriságot!

```
SortedMap<Character,Integer> map =
       new TreeMap<>();
Reader r = new InputStreamReader(System.in);
while (true) {
       int i = r.read();
       if (i < 0) break;
       char c = (char)i;
       int n = 0;
       if (map.containsKey(c)) n = map.get(c);
       n++;
       map.put(c, n);
}
for (char c : map.keySet()) {
       System.out.println("Char: " +c+"= \t"+map.get(c));
}
```

J2SE7

dobozol

`map.get(c)++` nem jó

végig a kulcsokon

For-each vagy Iterator?

- For-each ciklus
  - + olvashatóbb kód
  - - kollekció nem módosítható közben
- Iterator
  - + elemek közben törölhetők
  - - kód részletesebb, kevésbé olvasható(?)

`Collections` segédosztály

- Rendezés, min-max kiválasztás stb.
  - `sort(List<T> l)`
  - `sort(List<T> l, Comparator<T> c)`
- Megfordítás
  - `reverse(List<T> l)`
- Rotálás
  - `rotate(List<T> l, int distance)`
- Keverés
  - `shuffle(List<T> l)`
  - `shuffle(List<T> l, Random r)`

Kollekció példa

- Betűgyakoriság a gyakoriság sorrendjében

```
... // reading chars like previously
class Cmp implements
Comparator<SortedMap.Entry<Character, Integer> > {
    public int compare(Entry<Character, Integer> e1,
                       Entry<Character, Integer> e2) {
        return e2.getValue()-e1.getValue();
  }
}
List<SortedMap.Entry<Character, Integer>> l =
    new ArrayList<>();
l.addAll(map.entrySet());
Collections.sort(l, new Cmp());
for (SortedMap.Entry<Character, Integer> e : l) {
    System.out.println(e.getKey()+"= "+e.getValue());
}
```

comparator

segédlista

lista init

rendezés

Kollekciók megvédése

- Kollekciók paraméterként átadva
  - Java: referenciaként → alapból változtathatóak
  - C++: *const* kulcsszó → nem módosíthatók
- Hogyan adjuk át nem módosíthatóan Java-ban?
  - Védőcsomagolásban!
- **`java.util.Collections`**
  - **`Collection unmodifiableCollection(Collection c)`**
  - **`List unmodifiableList(List c)`**
  - **`Map unmodifiableMap(Map c)`**
  - **`Set unmodifiableSet(Set c)`**
  - generikus típusokkal is helyesen működik

Példa: nem módosítható lista

```
// kapunk egy listát valahonnan
List<Student> l = database.getStudents();

// továbbadjuk valami külső metódusnak:
// mi lesz a tartalmával?
doSomethingWithList(l);
```

```
// kapunk egy listát valahonnan
List<Student> l = database.getStudents();

// becsomagoljuk egy nem módosítható listába
// nem kell másolgatni, genericitás megmarad
List<Student> l2 = Collections.unmodifiableList(l);

// a csomagolót adjuk át, amiben a módosító metódusok
// UnsupportedOperationException-t dobnak ☺
checkForSomething(l2);
```


Köszönöm a figyelmet!