Genericitate

Programare Orientată pe Obiecte

Introducere

- concept nou - JDK 5.0.
- oferă un mijloc de **abstractizare a tipurilor de date**
- util mai ales în ierarhia de colecții
- din unele puncte de vedere, se poate asemăna cu conceptul de *template* din C++.

Fără genericitate

Exemplu:

```
List myList = new ArrayList();
myList.add(new Integer(0));
Integer x = (Integer) myList.get(0);
```

Obs: necesitatea operației de cast pentru a identifica corect variabila obținută din listă.

Dezavantaje:

- Este îngreunată citirea codului
- Apare posibilitatea unor erori la execuție
  - în momentul în care în listă se introduce un obiect care nu este de tipul *Integer*.

- *Genericitatea* intervine pentru a elimina aceste probleme!

Tipuri generice

- **Tipizarea elementelor unei colecții:**
  TipColectie < TipDeDate >

```
// Inainte de 1.5
    List list = new ArrayList();
    list.add(new Integer(123));
    int val = ((Integer)list.get(0)).intValue();

// Dupa 1.5, folosind si autoboxing
    List<Integer> list = new ArrayList<Integer>();
    list.add(123);
    int val = list.get(0);
```

- **Avantaje**: simplitate, control (eroare la compilare vs. ClassCastException)

Exemplu cu genericitate

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(new Integer(0));
Integer x = myList.get(0);
```

- lista nu mai conține obiecte oarecare, ci poate conține doar obiecte de tipul *Integer*.
- a dispărut şi cast-ul.
- **verificarea tipurilor este efectuată de compilator**, ceea ce elimină potențialele erori de execuție cauzate de eventuale cast-uri incorecte.

Beneficiile utilizării genericității:

- îmbunătățirea lizibilității codului
- creşterea gradului de robustețe

Observații

- Deoarece colecțiile sunt construite peste tipul de date Object, metodele de tip *next* sau *prev* ale iteratorilor vor returna tipul Object, fiind responsabilitatea noastră de a face conversie la alte tipuri de date, dacă e cazul!

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

sau:

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

sau:

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

Exemple

```
Collection<String> c = new ArrayList<String>();
c.add("Test");
c.add(2); // EROARE!
Iterator<String> it = c.iterator();
while (it.hasNext()) {
    String s = it.next();
}
```

- O **iterare** obişnuită pe un map se va face în felul următor:

```
for (Map.Entry<String,Student> entry:
    students.entrySet())
    System.out.println("Media “+ entry.getKey()+“este“
    +entry.getValue().getAverage());
```

- bucla for-each⇔ iteratorul mulțimii de perechi întoarse de entrySet.

Definirea unor structuri generice simple

- Exemple - definiția oferită de Java pentru tipurile *List* şi *Iterator*.

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

- Sintaxa <E> - folosită pentru a defini ***tipuri formale*** în cadrul interfețelor.
- În momentul în care invocăm efectiv o structură generică, ele vor fi înlocuite cu tipurile efective utilizate în învocare.

Definirea unor structuri generice simple

Exemplu:

```
ArrayList<Integer> myList = new
    ArrayList<Integer>();
Iterator<Integer> it = myList.iterator();
```

- În această situație, tipul formal *E* a fost înlocuit (la compilare) cu tipul efectiv *Integer*.

- Observație: analogie cu parametrii funcțiilor: se definesc utilizând parametri *formali*, urmând ca, în momentul unui apel, aceşti parametri să fie înlocuiți cu parametri *actuali*.

Genericitatea în subtipuri

Exemplu:

```
List<String> stringList = new ArrayList<String>();
                                        // corect!
List<Object> objectList = stringList;
                                        // operație corectă?
```

Presupunem că operația e corectă =>

- am putea introduce în *objectList* orice fel de obiect, nu doar obiecte de tip String =>
- Potențiale erori de execuție.
- Exemplu

```
objectList.add(new Object());
String s = stringList.get(0); // operatie ilegala!
```

=> Operația **nu va fi permisă de către compilator**!

Observație importantă

SubType este un subtip

- (clasă descendentă sau
- subinterfață)

al lui *SuperType*

Atenție!

O structură generică:

GenericStructure <SubType>

NU este un subtip al lui:

GenericStructure < SuperType >!

Wildcards

- Utilizate atunci când dorim să intrebuințăm o structură generică drept *parametru* într-o funcție şi nu dorim să limităm tipul de date din colecția respectivă.
- Exemplu fără wildcard:

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

- ne restricționează să folosim la apelul funcției doar o colecție cu elemente de tip *Object* (care ***nu*** *poate fi convertită la o colecție de un alt tip*)!

Wildcards

- Această restricție este eliminată de folosirea **wildcard**-urilor:
- Exemplu:

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

- Limitare: **nu putem adăuga elemente arbitrare** într-o colecție cu wildcard-uri:

```
Collection<?> c = new ArrayList<String>();
                        // Operatie permisa
c.add(new Object());
                        // Eroare la compilare
```

- De ce?

Wildcards

- Nu putem adăuga într-o colecție generică decât elemente **de un anumit tip**, iar wildcard-ul **nu indică un tip anume!**
- Putem adăuga elemente de tip *String?*

```
List<?> myList = new ArrayList<String>();
myList.add("Some String");
    // Eroare compilare!!
```

- Singurul element care poate fi adăugat este *null*, întrucât acesta este membru al oricărui tip referință!

- Rezolvare:

```
List<?> someList = new ArrayList<String>();
((ArrayList<String>)someList).add("Some String");
```

- Operațiile de tip *get* sunt posibile
  - rezultatul acestora poate fi mereu interpretat drept *Object*:

```
Object item = someList.get(0);
```

Exemple

- Corect? Dacă da, ce afişează?

```
List<?> someList = new ArrayList<String>();
((ArrayList<Integer>)someList).add(1);
System.out.print(item);
```

- Corect? Dacă da, ce afişează?

```
List<?> someList = new ArrayList<String>();
((ArrayList<String>)someList).add("Some String");
((ArrayList<Integer>)someList).add(1);
Object item = someList.get(0);
System.out.println(item);
item = someList.get(1);
System.out.println(item);
```

Bounded Wildcards

- un *wildcard* poate fi înlocuit cu orice tip => poate deveni un inconvenient!

Mecanismul bazat pe **Bounded Wildcards:**

- permite introducerea unor restricții asupra tipurilor ce pot inlocui un wildcard
- le obligă să se afle într-o relație ierarhică (de moştenire) față de un tip fix specificat.

*Clasă(sau interfață) <? **extends** Bază>*

- impune ca tipul elementelor clasei (interfeței) să fie de tip **Bază** sau un **subtip** al acesteia!

*Clasă(sau interfață) <? **super** Bază>*

- impune ca tipul elementelor listei să fie de tip **Bază** sau o **superclasă** a acesteia

Exemplu

```
class User{
    protected String name = "User";
    public String getName() {
        return name;
    }
}
class Admin extends User {
    public Admin () {
        name = “Admin ";
    }
}
class Student extends User {
    public Student () {
        name = " Student ";
    }
}
```

Exemplu

```
class MyApplication {
    // bounded wildcards
    public static void listUser(List<? extends User> userList) {
        for(User item : userList)
            System.out.println(item.getName());
    }

    public static void main(String[] args) {
        List<User> pList = new ArrayList<User>();

        pList.add(new Admin());
        pList.add(new Student());
        pList.add(new User());

        MyApplication.listUser(pList);
        // Se va afisa: “Admin", “Student", "User"
    }
}
```

Metode generice

- au un tip-parametru pentru a facilita prelucrarea unor structuri generice (date ca parametru).
- Exemple de implementare ale unei metode ce copiază elementele unui vector intrinsec într-o colecție:
- Metodă corectă

```
static <T> void correctCopy(T[] a, Collection<T> c){
    for (T o : a) c.add(o); // Operatia va fi permisa
}
```

- Metodă incorectă

```
static void incorrectCopy(Object[] a, Collection<?> c){
    for (Object o : a) c.add(o);
/* Operatie incorecta, eroare la compilare: adăugarea
   elementelor într-o colecție generică cu tip
   specificat*/
}
```

Metode generice

- putem folosi *wildcards* sau *bounded wildcards*.

- Exemple corecte:
- metodă ce copiaza elementele dintr-o listă în altă listă

```
public static <T> void
        copy(List<T> dest, List<? extends T> src) { …}
```

- metodă de adăugare a unor elemente într-o colecție, cu restricționarea tipului generic

```
public boolean addAll (int index,Collection<? extends
    E> c)
```

Tipuri de date enumerare

```
enum

public class CuloriSemafor {
      public static final int ROSU = -1;
      public static final int GALBEN = 0;
      public static final int VERDE = 1;
}

...
// Exemplu de utilizare
if (semafor.culoare == CuloriSemafor.ROSU)
semafor.culoare = CuloriSemafor.GALBEN;

// Doar de la versiunea 1.5 !

public enum CuloriSemafor { ROSU, GALBEN, VERDE };
// Utilizarea structurii se face la fel
```