Genericitatea

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

Introducere

Genericitatea este un concept nou, introdus o data cu JDK 5.0. Din unele puncte de vedere, se poate asemana cu conceptul de ***template*** din C++. Mecanismul genericitatii ofera un mijloc de abstractizare a tipurilor de date si este util mai ales in ierarhia de **colectii**.

Sa urmarim urmatorul exemplu:

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

Se observa necesitatea operatiei de *cast* pentru a identifica corect variabila obtinuta din lista. Aceasta situatie are mai multe **dezavantaje**:

- Este **ingreunata** citirea codului
- Apare posibilitatea unor **erori** la executie, in momentul in care in lista se introduce un obiect care nu este de tipul `Integer`.

Genericitatea intervine tocmai pentru a elimina aceste probleme. Concret, sa urmarim secventa de cod de mai jos:

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

In aceasta situatie, lista nu mai contine obiecte oarecare, ci poate contine **doar** obiecte de tipul `Integer`. In plus, observam ca a disparut si *cast*-ul. De aceasta data, verificarea tipurilor este efectuata de **compilator**, ceea ce elimina potentialele erori de executie cauzate de eventuale *cast*-uri incorecte. La modul general, **beneficiile** dobandite prin utilizarea genericitatii constau in:

- imbunatatirea **lizibilitatii** codului
- cresterea gradului de **robustete**.

Definirea unor structuri generice simple

Sa urmarim cateva elemente din definitia oferita de Java pentru tipurile `List` si `Iterator`.

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

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

Sintaxa `<E>` este folosita pentru a defini **tipuri formale** in cadrul interfetelor. Aceste tipuri pot fi folosite in mod asemanator cu tipurile uzuale (cu anumite restrictii totusi). In momentul in care invocam efectiv o structura generica, ele vor fi inlocuite cu **tipurile efective** utilizate in invocare. Concret, fie un apel de forma:

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

In aceasta situatie, tipul formal `E` a fost inlocuit cu tipul efectiv `Integer`.

Observatie: O analogie (simplista) referitoare la acest mecanism de lucru cu tipurile se poate face cu mecanismul functiilor: acestea se definesc utilizand **parametri formali**, urmand ca, in momentul unui apel, acesti parametrii sa fie inlocuiti cu **parametri actuali**.

Genericitatea in subtipuri

Sa consideram urmatoarea situatie:

```
List<String> stringList = new ArrayList<String>(); // 1
List<Object> objectList = stringList;               // 2
```

Operatia 1 este evident corecta, insa este corecta si operatia 2? Presupunand ca ar fi, am putea introduce in `objectList` orice fel de obiect, nu doar obiecte de tip `String`, fapt ce ar conduce la potentiale erori de executie, astfel:

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

Din acest motiv, operatia 2 **nu** va fi permisa de catre compilator! Generalizand, daca `ChildType` este un subtip (clasa descendenta sau subinterfata) al lui `ParentType`, atunci o structura generica `GenericStructure<ChildType>` **NU** este un subtip al lui `GenericStructure<ParentType>`. **Atentie** la acest concept, intrucat el nu este intuitiv!

Wildcards

Wildcard-urile sunt utilizate atunci cand dorim sa intrebuintam o structura generica drept parametru intr-o functie si **nu** dorim sa **limitam** tipul de date din colectia respectiva. De exemplu, o situatie precum urmatoarea ne-ar restrictiona sa folosim la apelul functiei doar o colectie cu elemente de tip `Object` (ceea ce **NU** poate fi convertita la o colectie de un alt tip, dupa cum am vazut mai sus):

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

Aceasta restrictie este eliminata de folosirea *wildcard*-urilor, dupa cum se poate vedea:

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

O **limitare** care intervine insa este ca **nu putem adauga** elemente arbitrare intr-o colectie cu *wildcard*-uri:

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

Eroarea apare deoarece nu putem adauga intr-o colectie generica decat elemente de un anumit tip, iar wildcard-ul nu indica un tip anume.

Observatie: Aceasta inseamna ca nu putem adauga nici macar elemente de tip `String`. **Singurul** element care poate fi adaugat este insa `null`, intrucat acesta este membru al oricarui tip referinta. Pe de alta parte, operatiile de tip *getter* sunt posibile, intrucat rezultatul acestora poate fi mereu interpretat drept `Object`:

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

Bounded Wildcards

In anumite situatii, faptul ca un wildcard poate fi inlocuit cu **orice** tip se poate dovedi un **inconvenient**. Mecanismul bazat pe ***Bounded Wildcards*** permite introducerea unor restrictii asupra tipurilor ce pot inlocui un *wildcard*, obligandu-le sa se afle intr-o relatie **ierarhica** fata de un tip fix specificat.

Exemplificam acest mecanism:

```
class Pizza {
    protected String name = "Pizza";
```

```
    public String getName() {
        return name;
    }
}

class HamPizza extends Pizza {
    public HamPizza() {
        name = "HamPizza";
    }
}

class CheesePizza extends Pizza {
    public CheesePizza() {
        name = "CheesePizza";
    }
}

class MyApplication {
    // Aici folosim "bounded wildcards"
    public static void listPizza(List<? extends Pizza> pizzaList) {
        for(Pizza item : pizzaList)
            System.out.println(item.getName());
    }

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

        pList.add(new HamPizza());
        pList.add(new CheesePizza());
        pList.add(new Pizza());

        MyApplication.listPizza(pList);
        // Se va afisa: "HamPizza", "CheesePizza", "Pizza"
    }
}
```

Sintaxa `List<? extends Pizza>` impune ca tipul elementelor listei sa fie `Pizza` sau o **subclasa** a acesteia. Astfel, `pList` ar fi putut avea, la fel de bine, tipul `List<HamPizza>` sau `List<CheesePizza>`. In mod similar, putem imprima constrangerea ca tipul elementelor listei sa fie Pizza sau o **superclasa** a acesteia, utilizand sintaxa `List<? super Pizza>`.

Observatie: Trebuie retinut faptul ca in continuare nu putem introduce valori intr-o colectie ce foloseste *bounded wildcards* si este data ca parametru unei functii.

Spre deosebire de

Metode generice

Java ne ofera posibilitatea scrierii de metode generice (deci avand un tip-parametru) pentru a facilita prelucrarea unor structuri generice (date ca parametru).

Sa exemplificam acest fapt. Observam in continuare 2 cai de implementare ale unei metode ce copiaza elementele unui vector intrinsec intr-o colectie:

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

// Metoda incorecta
static void incorrectCopy(Object[] a, Collection<?> c) {
    for (Object o : a)
        c.add(o); // Operatie incorecta, semnalata ca eroare de catre compilator
}
```

Trebuie remarcat faptul ca `correctCopy()` este o metoda valida, care se executa corect, insa `incorrectCopy()` **nu** este, din cauza limitarii pe care o cunoastem deja, referitoare la adaugarea elementelor intr-o colectie generica cu tip specificat. Putem remarca, de asemenea, ca, si in acest caz, putem folosi *wildcards* sau *bounded wildcards*. Astfel, urmatoarele declaratii de metode sunt corecte:

```
// O metoda ce copiaza elementele dintr-o lista in alta lista
public static <T> void copy(List<T> dest, List<? extends T> src) { ... }

// O metoda de adaugare a unor elemente intr-o colectie, cu restrictionarea tipului generic
public <T extends E> boolean addAll(Collection<T> c);
```

Exercitii

1. (**2p**) Realizati un **test de performanta** in care veti compara o colectie **generica** de valori intregi cu una **non-generica**. Pentru aceasta veti masura, in ambele situatii, intervalul necesar introducerii si apoi al accesarii tuturor elementelor din fiecare lista (o accesare presupune returnarea valorii intregi continuta intr-o anumita pozitie). Explicati rezultatele obtinute.
2. (**2p**) Implementati o **coada de prioritati** care accepta doar tipuri numerice (descendente din `java.lang.Number`). Folositi coada pentru a **sorta** in ordine crescatoare o serie de valori numerice (generate aleator). Valorile numerice nu se vor stoca in vectori sau colectii ajutatoare. Veti utiliza *bounded wildcards*.
3. (**2p**) Modificati exercitiul precedent astfel:
   - Folositi o **colectie generica** in care veti stoca valorile numerice initiale
   - Implementati o **metoda generica** de copiere a valorilor din aceasta colectie in coada de prioritati.
4. (**5p**) Se considera interfata `Sumabil`, ce contine metoda:

   ```
   void addValue(Sumabil value)
   ```

   Aceasta metoda aduna la valoarea curenta (stocata in instanta ce apeleaza metoda) o alta valoare, aflata intr-o instanta cu acelasi tip. Pornind de la aceata interfata, va trebui sa:
   - Definiti **clasele** `MyVector3` si `MyMatrix` (ce reprezinta un vector cu 3 coordonate si o matrice de dimensiune 4 x 4), ce implementeaza `Sumabil`
   - Scrieti o **metoda generica** ce primeste o colectie generica cu elemente de tipul `Sumabil` si returneaza suma tuturor elementelor din colectie. Trebuie sa utilizati *bounded types*. Cu alte cuvinte, antetul functiei trebuie sa fie:

     ```
     static <T extends Sumabil> T addAll( ... )
     ```

- **Solutii**

Adus de la "http://cursuri.cs.pub.ro/~poo/wiki/index.php/Genericitatea"