Programare orientată pe obiecte

Breviar - Laboratorul 9

Mihai Nan
mihai.nan.cti@gmail.com

Facultatea de Automatică şi Calculatoare
Universitatea Politehnica din Bucureşti
Anul universitar 2015 - 2016

1 Genericitate

1.1 Introducere

Am prezentat in laboratorul anterior principalele avantaje ale utilizarii mecanismului de abstractizare a tipurilor de date in limbajul Java. In cadrul acestui laborator, vom analiza in detaliu acest mecanism. Tipurile generice simplifica lucrul cu colectii, permitand tipizarea elementelor acestora. Definirea unui tip generic se realizeaza prin specificarea intre paranteze unghiulare a unui tip de date Java, efectul fiind impunerea tipului respectiv pentru toate elementele colectiei.

⚠ IMPORTANT !

⚠ In cazul folosirii tipurilor generice, incercarea de a utiliza in cadrul unei colectii a unui element necorespunzator ca tip va produce o eroare la compilare, spre deosebire de varianta in care tipurile generice nu sunt uzitate, ce permite doar aruncarea unor exceptii de tipul ***ClassCastException***, in cazul folosirii incorecte a tipurilor.

1.2 Definirea structurilor generice

Pentru o intelegere mai buna a conceptului de structura generica, pornim de la clasa definita mai jos.

Cod sursa Java

```
class Association<K, V> {
    private K key;
    private V value;

    public Association(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return this.key;
    }

    public V getValue() {
        return this.value;
    }
}
```

Sintaxa ***<K, V>*** este folosita pentru a defini **tipuri formale** in cadrul definitiei clasei. Aceste tipuri pot fi folosite in mod asemanator cu tipurile uzuale. In momentul in care invocam efectiv o structura generica, ele vor fi inlocuite cu **tipurile efective**, utilizate in invocare. Astfel, pentru a instantia obiecte de tipul Assocition, folosim apeluri de forma:

Cod sursa Java

```
class Test {
    public static void main(String args[]) {
        Association<String, String> map1;
        map1 = new Association<String, String>("CC", "POO");
        String s1 = map1.getKey();
        Association<String, Integer> map2;
        map2 = new Association<String, Integer>("POO", 2015);
        String s2 = map2.getKey();
        int nr = map2.getValue();
    }
}
```

Se poate observa ca, in aceste exemple, **tipurile formale**, ***K*** si ***V***, au fost inlocuite cu **tipuri efective** (***String***, ***Integer***). De asemenea, de remarcat este si faptul ca ambele metode de tip ***get***, din implementarea clasei analizate, au ca rezultat un obiect cu tip formal, iar in cadrul apelului nu a fost nevoie de operatii de tip ***cast***.

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**.

1.2.1 Genericitatea in subtipuri

Consideram, spre analiza, urmatoarea situatie, descrisa prin codul de mai jos. Aceasta situatie contine doua operatii, iar despre prima stim cu siguranta ca este una valida, fiind explicata anterior. Ne intereseaza daca a doua operatie este posibila si o explicatie a raspunsului la aceasta intrebare.

Cod sursa Java

```
class Test {
    public static void main(String args[]) {
        Association<String, Integer> map1;
        //Operatia 1
        map1 = new Association<String, Integer>("POO", 2015);
        Association<Object, Object> map2;
        //Operatia 2
        map2 = map1;
    }
}
```

Presupunand, prin absurd, ca operatia 2 este una corecta, am putea introduce in ***map2*** orice fel de obiecte, fapt ce ar conduce la potentiale erori de executie. Din acest motiv, operatia 2 nu va fi permisa de catre compilator!

⚠ IMPORTANT !

⚠ Generalizand, daca ***CType*** este un subtip (clasa descendenta sau subinterfata) al lui ***PType***, atunci o structura generica ***GenericStructure<CType>*** **NU** este un subtip al lui ***GenericStructure<PType>***. **Atentie** la acest concept, intrucat el nu este intuitiv!

1.2.2 Restrictionarea parametrilor

Exista cazuri in care dorim sa adaugam intr-o structura de date generica elemente care au un tip cu anumite proprietati. De cele mai multe ori, acest lucru se intampla in cazul structurilor generice care contin elemente cu tipuri derivate. Spre exemplu, sa spunem ca dorim sa realizam o clasa ***AVector*** care extinde clasa ***AbstractList*** si vrem ca elementele din structura sa fie subtipuri ale lui ***Number***.

Cod sursa Java

```
abstract class AVector<E extends Number> extends
        AbstractList<E> {
    abstract public boolean add(E obj);
    abstract public E get(int index);
    abstract public Enumeration<E> elements();
    abstract public Iterator<E> iterator();
    abstract public ListIterator<E> listIterator();
}
```

Sintaxa ***<E extends Number>*** indica faptul ca tipul ***E*** este o subclasa a lui ***Number*** (sau chiar ***Number***). Aceasta restrictie face imposibila instantierea unui obiect, avand un tip derivat din ***AVector***, care sa contina elemente de tip ***String***, deoarece ***String*** nu este o subclasa a lui ***Number***.

Fie clasa ***Vector*** o clasa instantiabila care extinde ***AVector***. Dupa ce am realizat instantierea unui obiect ***Vector***, precizand ca tip pentru elementele acestui ***Integer***, nu mai putem adauga in aceste elemente de tip ***Double***, spre exemplu, chiar daca, la o prima vedere, acest lucru ar fi posibil, deoarece ***Double*** reprezinta un subtip al lui *Number*. Pentru o intelegere mai buna a celor expuse, este recomandata analizarea exemplului de cod propus mai jos.

Cuvantul cheie ***extends*** este folosit si in cazul in care dorim sa indicam un tip ce implementeaza o anumita interfata. Un exemplu in acest sens poate fi clasa definita in blocul de cod urmator.

Cod sursa Java

```
class Map<K extends Number, V extends Set<K>>
        extends Association<K, V> {
    public Map(K key, V value) {
        super(key, value);
    }

    public static void main(String args[]) {
        Map<Double, TreeSet<Double>> obj;
        TreeSet<Double> v = new TreeSet<Double>();
        v.add(new Double(2.5));
        obj = new Map<Double, TreeSet<Double>>(1.2, v);
        Double x = obj.getKey();
        System.out.println(x);
        TreeSet<Double> set = obj.getValue();
        System.out.println(set);
    }
}
```

1.3 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):

Cod sursa Java

```
void boolean containsAll(Collection<Object> c) {
    for (Object e : c) {
        if(!this.contains(e)) {
            return false;
        }
    }
    return true;
}
```

Aceasta restrictie este eliminata de folosirea **wildcard-urilor**, dupa cum se poate vedea in blocul de cod de mai jos.

Cod sursa Java

```
void boolean containsAll(Collection<?> c) {
    for (Object e : c) {
        if(!this.contains(e)) {
            return false;
        }
    }
    return true;
}
```

⚠ IMPORTANT !

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

Cod sursa Java

```
class TestClass {
    public static void main(String args[]) {
        //Operatie permisa
        Collection<?> c = new LinkedList<Integer>();
        //Eroare la compilare
        c.add(new Object());
    }
}
```

Observatie

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

Cod sursa Java

```
class TestClass {
    public static void main(String args[]) {
        Set<?> set = new HashSet<Integer>();
        ((HashSet<Integer>) set).add(10);
        Object item = set.iterator().hasNext();
        List<?> list = new LinkedList<String>();
        ((LinkedList<String>) list).add("POO");
        item = list.get(0);
    }
}
```

1.4 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.

Sintaxa ***Set<? extends Number>*** impune ca tipul elementelor multimii sa fie ***Number*** sau o subclasa a acesteia. Astfel, in exemplul de mai jos, obiectul ***set*** ar fi putut avea, la fel de bine, tipul ***HashSet<Short>*** sau ***TreeSet<Long>***. In mod similar, putem imprima constrangerea ca tipul elementelor multimii sa fie ***AbstractMethodError*** sau o superclasa a acesteia, utilizand sintaxa ***Set<? super AbstractMethodError>***.

⚠ IMPORTANT !

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

Cod sursa Java

```
class TestClass {
    public static void printType(Set<? extends Number> set) {
        for(Number item : set) {
            System.out.println(item.getClass());
        }
    }

    public static void main(String args[]) {
        Set<Number> set = new HashSet<Number>();
        set.add(new Integer(5));
        set.add(new Double(7.2));
        set.add(new Float(10.5));
        TestClass.printType(set);
    }
}
```

1.5 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), ceea ce am vazut si in subsectiunea anterioara.

Pentru o intelegere mai buna a acestui concept, se recomanda parcurgerea celor doua metode propuse, in vederea analizarii corectitudinii lor.

Cod sursa Java

```
class TestClass {
    //Metoda1
    public <T> void metoda1(T[] a, Collection<T> c) {
        for(T o : a) {
            //Operatie permisa
            c.add(o);
        }
    }

    //Metoda 2
    public void metoda2(Object[] a, Collection<?> c) {
        for(Object o : a) {
            //Eroare la compilare
            c.add(o);
        }
    }
}
```

Trebuie remarcat faptul ca ***metoda1*** este o metoda valida, care se excuta corect, insa nu putem spune acelasi lucru si despre ***metoda2***, din cauza limitarii, pe care am explicat-o deja in subsectiunea anterioara, referitoare la adaugarea elementelor intr-o colectie generica, avand un tip specificat. Este de remarcat faptul ca, in acest caz, putem folosi **wildcards**, dar si **bounded wildcards**. Acest lucru este evidentiat in blocul de cod urmator.

Cod sursa Java

```
class TestClass {
    List<Number> dest;

    public <T> void copy(Set<T> dest, List<? extends T> src) {
        for(T o : src) {
            dest.add(o);
        }
    }

    public <T extends Number> boolean addAll(Collection<T> c)
    {
        for(T o : c) {
            dest.add(o);
        }
        return true;
    }
}
```