Laboratorul 10: Genericitate

Obiective

Scopul acestui laborator este prezentarea conceptului de genericitate și modalitățile de creare și folosire a claselor, metodelor și interfețelor generice în Java.

Aspectele urmărite sunt:

- prezentarea structurilor generice simple
- conceptele de wildcard și bounded wildcards
- utilitatea genericității în design-ul unui sistem

Introducere

Să urmărim exemplul de mai jos:

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

Se observă necesitatea operației de *cast* pentru a identifica corect variabila obținută din listă. Această situație are mai multe 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 tocmai pentru a elimina aceste probleme. Concret, să urmărim secvența de cod de mai jos:

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

În această situație, lista nu mai conține obiecte oarecare, ci poate conține doar obiecte de tipul `Integer`. În plus, observăm că a dispărut și *cast*-ul. De această dată, **verificarea tipurilor este efectuată de compilator**, ceea ce elimină potențialele erori de execuție cauzate de *cast*-uri incorecte. La modul general, beneficiile dobândite prin utilizarea genericității constau în:

- îmbunătățirea lizibilității codului
- creșterea gradului de robusteţe

Definirea unor structuri generice simple

Să urmărim câteva elemente din definiția oferită de Java pentru tipurile `List` și `Iterator`.

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

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

Sintaxa `<E>` (poate fi folosită orice literă) este folosită pentru a defini tipuri formale în cadrul interfețelor. Aceste tipuri pot fi folosite în mod asemănător cu tipurile uzuale, cu anumite restricții totuși. În momentul în care invocăm o structură generică ele vor fi înlocuite cu tipurile efective utilizate în invocare. Concret, fie un apel de forma:

```
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`.

Genericitatea în subtipuri

Să considerăm următoarea situaţie:

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

Operația 1 este evident corectă, însă este corectă și operația 2? Presupunând că ar fi, am putea introduce în `objectList` orice fel de obiect, nu doar obiecte de tip `String`, fapt ce ar conduce la potențiale erori de execuție, astfel:

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

Din acest motiv, operația 2 **nu va fi permisă de către compilator**!

Dacă *ChildType* este un subtip (clasă descendentă sau subinterfață) al lui *ParentType*, atunci o structură generică *GenericStructure<ChildType>* **nu** este un subtip al lui *GenericStructure<ParentType>*. **Atenție** la acest concept, întrucât el nu este intuitiv!

Wildcards

Wildcard-urile sunt utilizate atunci când dorim să întrebuințăm o structură generică drept *parametru* într-o funcție și nu dorim să limităm tipul de date din colecția respectivă.

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

De exemplu, o situație precum cea de mai sus ne-ar restricționa să folosim la apelul funcției doar o colecţie cu elemente de tip `Object`, care **nu** *poate fi convertită la o colecție de un alt tip*, după cum am văzut mai sus. Această restricție este eliminată de folosirea **wildcard**-urilor, după cum se poate vedea:

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

O limitare care intervine însă este că **nu putem adăuga elemente arbitrare** într-o colecție cu *wildcard*-uri:

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

Eroarea apare deoarece nu putem adăuga într-o colecţie generică decât elemente **de un anumit tip**, iar *wildcard*-ul **nu indică un tip anume**.

Aceasta înseamnă că nu putem adăuga nici măcar elemente de tip `String`. Singurul element care poate fi adăugat este însă `null`, întrucât acesta este membru al oricărui tip referință. Pe de altă parte, operațiile de tip *getter* sunt posibile, întrucât 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

În anumite situații, faptul că un *wildcard* poate fi înlocuit cu orice tip se poate dovedi un inconvenient. Mecanismul bazat pe **Bounded Wildcards** permite introducerea unor restricţii asupra tipurilor ce pot înlocui un *wildcard*, obligându-le să se afle într-o relație ierarhică (de descendență) față de un tip fix specificat.

Exemplificăm 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 afişa: "HamPizza", "CheesePizza", "Pizza"
    }
}
```

Sintaxa `List<? extends Pizza>` (Upper Bounded Wildcards [https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html]) impune ca tipul elementelor listei să fie `Pizza` sau o subclasă a acesteia. Astfel, `pList` ar fi putut avea, la fel de bine, tipul `List<HamPizza>` sau `List<CheesePizza>`. În mod similar, putem imprima constrângerea ca tipul elementelor listei să fie `Pizza` sau o superclasă a acesteia, utilizând sintaxa `List<? super Pizza>` (Lower Bounded Wildcards [https://docs.oracle.com/javase/tutorial/java/generics/lowerBounded.html]).

Utilizarea bounded wildcards se manifestă în următoarele 2 situații :

- lower bounded wildcards se folosesc atunci când vrem să **modificăm** o colecție generică
- upper bounded wildcards se folosesc atunci când vrem să **parcurgem fără să modificăm** o colecție generică

Type Erasure

Type Erasure [https://docs.oracle.com/javase/tutorial/java/generics/erasure.html] este un mecanism prin care compilatorul Java înlocuieşte la **compile time** parametrii de genericitate ai unei clase generice cu prima lor apariţie (ţinând cont de restricţii în cazul Bounded Wildcards) sau cu `Object` dacă parametrii nu apar (Raw Type). De exemplu, următorul cod:

```
List<String> list = new ArrayList<String>();
list.add("foo");
String x = list.get(0);
```

se va transforma după acest pas al compilării în:

```
List list = new ArrayList();
list.add("foo");
String x = (String) list.get(0);
```

Să urmărim următorul fragment de cod:

```
class  GenericClass <T> {
    void genericFunction(List<String> stringList) {
        stringList.add("foo");
    }
    // {...}
    public static void main(String[] args) {
        GenericClass genericClass = new GenericClass();
        List<Integer> integerList = new ArrayList<Integer>();

        integerList.add(100);
        genericClass.genericFunction(integerList);

        System.out.println(integerList.get(0)); // 100
        System.out.println(integerList.get(1)); // foo
    }
}
```

Observăm că în `main` se instanţiază clasa `GenericClass` cu Raw Type, apoi se trimite ca argument metodei `genericFunction` un `ArrayList<Integer>`. Codul nu va genera erori şi va afişa *100*, apoi *foo*. Acest lucru se întâmplă tot din cauza mecanismului de **Type Erasure**. Să urmărim ce se întâmplă: la instanţierea clasei `GenericClass` nu se specifică tipul generic al acesteia iar compilatorul va înlocui în corpul clasei peste tot `T` cu `Object` şi va dezactiva verificarea de tip. Aşadar, obiectul `genericClass` va aparţine unei clase de forma:

```
class  GenericClass {
    void genericFunction(List stringList) {
        stringList.add("foo");
    }
    // {...}
}
```

Modelul de mai sus este **bad practice** tocmai pentru că are un comportament nedeterminat și poate conduce la erori. De aceea nu e recomandat să folosiți Raw Types, ci să specificați **întotdeauna** tipul obiectelor în cazul instanțierii claselor generice!

Metode generice

Java ne oferă posibilitatea scrierii de metode generice (deci având un tip-parametru) pentru a facilita prelucrarea unor structuri generice. Să exemplificăm acest fapt. Observăm în continuare 2 căi de implementare ale unei metode ce copiază elementele unui vector intrinsec într-o colecție:

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

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

Trebuie remarcat faptul că `correctCopy()` este o metodă validă, care se execută corect, însă `incorrectCopy()` nu este, din cauza limitării pe care o cunoaştem deja, referitoare la adăugarea elementelor într-o colecție generică cu tip specificat. Putem remarca, de asemenea, că, și în acest caz, putem folosi *wildcards* sau *bounded wildcards*. Astfel, următoarele declaraţii de metode sunt corecte:

```
// Copiază elementele dintr-o listă în altă listă
public static <T> void copy(List<T> dest, List<? extends T> src) { ... }

// Adaugă elemente dintr-o colecţie în alta, cu restricţionarea tipului generic
public <T extends E> boolean addAll(Collection<T> c);
```

Exemple genericitate

Probabil nu sunteți familiari încă cu termenul de “GPU Computing” (utilizarea unui procesor grafic pentru accelerarea calculelor), dar probabil ați exploatat una dintre întrebuințările ei, mai exact jocurile video.

Jocurile video sunt create cu ajutorul unor engine-uri grafice, care în esență nu reprezintă altceva decât aplicații care realizează o multitudine de operații matematice: plotări de grafice, interpolări, operații matriceale/vectoriale, derivări etc.

Aceste operații matematice pot fi făcute pe diferite tipuri de date. O matrice poate acceptă int-uri (exemplu: camera jucătorului), float-uri / double-uri (exemplu: setarea opacității unei texturi), char-uri (exemplu: reprezentarea text box-urilor pentru dialog) etc. În loc să creem câte o clasă care să adere fiecărui tip, putem scrie o singură dată o clasă care să reprezinte o matrice și care să accepte mai multe tipuri de date prin genericitate. Acest lucru devine foarte util dacă dorim să creem o bibliotecă întreagă pentru operații matematice avansate (exemplu: Jscience), fără să ne repetăm codul doar pentru a crea clase și metode specifice unor tipuri de date.

Exerciții

Scheletul laboratorului poate fi descărcat de aici: oop_lab10.zip

Laboratorul trebuie rezolvat pe platforma LambdaChecker, fiind găsit aici [https://beta.lambdachecker.io/contest/21].

1. **(6 puncte)** Implementați o structură de date de tipul `MultiMapValue<K, V>`, pe baza scheletului, care reprezintă un `HashMap<K, ArrayList<V>`, unde o cheie este asociată cu mai multe valori. Modalitatea de stocare a datelor este la alegere (moștenire sau agregare) și să folosiți funcționalitățile din HashMap. În schelet aveți următoarele metode de implementat:
    a. **(1 punct)** `add(K key, V value)` - adaugă o valoare la o cheie dată (valoarea este adăugate în lista de valori asociate cheii, dacă cheia și lista nu există, atunci lista va fi creată și asociată cheii.
    b. **(1 puncte)** `void addAll(K key, List<V> values)` - adaugă valorile din lista de valori dată ca parametru la lista asociată cheii.
    c. **(1 puncte)** `void addAll(MultiMapValue<K, V> map)` - adaugă intrările din obiectul MultiMapValue dat ca parametru în obiectul curent (this).
    d. **(0.5 puncte)** `V getFirst(K key)` - întoarce prima valoare asociată cheii (dacă nu există, se întoarce null).
    e. **(0.5 puncte)** `List<V> getValues(K key)` - se întoarce lista de valori asociată cheii.
    f. **(0.5 puncte)** `boolean containsKey(K key)` - se verifică faptul dacă este prezentă cheia în MultiMapValue.
    g. **(0.5 puncte)** `boolean isEmpty()` - se verifică dacă MultiMapValue este gol.
    h. **(0.5 puncte)** `List<V> remove(K key)` - se șterge cheia, împreună cu valorile asociate ei, din MultiMapValue.
    i. **(0.5 puncte)** `int size()` - se întoarce mărimea MultiMapValue.

1. **(4 puncte)** Implementați o structură de date de tipul Tree<T> (Arbore binar de căutare) pe baza scheletului. Analizați modalitatea de utilizare a bounded wildcards, explicați necesitatea lor laborantului (fie în cadrul orei de laborator, fie la nivel de comentariu în cod). În schelet aveți următoarele metode de implementat:
    a. **(1 puncte)** `void addValue(T value)` - adaugă o valoare în arborele binar de căutare.
    b. **(0.5 puncte)** `void addAll(List<T> values)` - adaugă valorile dintr-o listă în arborele binar de căutare.
    c. **(1.5 puncte)** `HashSet<T> getValues(T inf, T sup)` - colectează valorile din arbore între o limită inferioară și superioară într-o colecție de tipul HashSet.
    d. **(0.5 puncte)** `int size()` - se întoarce numărul de elemente inserate în arbore.
    e. **(0.5 puncte)** `boolean isEmpty()` - se întoarce dacă există vreun element inserat în arborele binar sau nu.

Referinţe

- Generic Types [https://docs.oracle.com/javase/tutorial/java/generics/types.html]
- Wildcards [https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html]
- Upper Bounded Wildcards [https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html]
- Lower Bounded Wildcards [https://docs.oracle.com/javase/tutorial/java/generics/lowerBounded.html]
- Type Erasure [https://docs.oracle.com/javase/tutorial/java/generics/erasure.html]

poo-ca-cd/laboratoare/genericitate.txt · Last modified: 2022/12/18 01:53 by cvintilian.rosca