# Lucrare de laborator Nr 7

# GENERICITATE

## Obiective

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

După studierea acestui laborator studenții vor fi capabili:

- să prezinte exemple de structuri generice simple;
- să utilizeze structuri generice în java code;
- să înțeleagă și să utilizeze corect conceptele de **wildcard** și **bounded wildcards**;
- să identifice și să explice utilitatea genericității în design-ul unui sistem.

## 1. 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 robustețe

## 2. 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**.

### 3. 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!

### 4. 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 colecție 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 colecție 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);
```

### 5. 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 restricții 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) 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).

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

### 6. Type Erasure

Type Erasure este un mecanism prin care compilatorul Java înlocuiește la **compile time** parametrii de genericitate ai unei clase generice cu prima lor apariție (ținând cont de restricții 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 și va afișa *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* și va dezactiva verificarea de tip. Așadar, obiectul *genericClass* va aparține 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!

### 7. 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 cunoaștem 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 declarații 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 colecție în alta, cu restricționarea tipului generic
public <T extends E> boolean addAll(Collection<T> c);
```

## EXERCIȚII

1. (**6p**) Implementați o **tabelă de dispersie** generică care va permite să stocați perechi de tip cheie-valoare.
   - (**2p**) Scrieți antetul clasei *MyHashMap* și prototipul funcțiilor **put** și **get**. Aveți grijă la parametrizarea tipurilor.
   - (**2p**) Implementați metoda **put**. Vă puteți crea o clasă internă cu rol de *entry* și puteți stoca *entry-urile* într-o colecție generică existentă în Java.
   - (**1p**) Implementați metoda **get**.
   - (**1p**) Testați implementarea voastră folosind o clasă definită de voi, care suprascrie metoda **hashCode** din *Object*.

2. (**4p**) Să considerăm interfața *Sumabil*, ce conține metoda *void addValue(Sumabil value)*. Această metodă adună la valoarea curentă (stocată în instanța ce apelează metoda) o altă valoare, aflată într-o instanță cu același tip. Pornind de la această interfață, va trebui să:
   - Definiți clasele *MyVector3* și *MyMatrix* (ce reprezintă un vector cu 3 coordonate și o matrice de dimensiune 4 x 4), ce implementează *Sumabil*
   - Scrieți **o metodă generică** ce primește o colecție generică cu elemente de tipul *Sumabil* și returnează suma tuturor elementelor din colecție. Trebuie să utilizați *bounded types*. Care trebuie să fie, deci, antetul metodei?

## Întrebări de control

1. Cum înțelegeți noțiunea de genericitate în java?
2. Care sunt avantajele utilizării construcțiilor generice în java?
3. Ce este un wildcard?
4. Pentru ce se utilizează wildcard-urile?
5. Ce este un bounded wildcard?
6. În ce cazuri se folosesc bounded wildcard?
7. Ce reprezintă mecanismul Type Erasure, cum se implementează?
8. Ce reprezintă metode generice, în ce cazuri se utilizează?

## Referinţe

- Generic Types
- Wildcards
- Upper Bounded Wildcards
- Lower Bounded Wildcards
- Type Erasure