**Laborator 7 POO**

***Genericitate***

Genericitatea face referire la tipuri de date parametrizate in Java. Ideea principala a acestui concept este de a permite unui tip de date (Integer, String, …, user-defined types) sa fie parametru pentru metode, clase sau interfate. Genericitatea este asemanatoare cu conceptul de template in C++. Spre exemplu, clase precum HashSet, ArrayList, HashMap, etc., utilizeaza tipuri de date generice. Motivul principal am introducerii tipurilor generice este ca superclasa Object poate face referire la orice tip de obiect, ceea ce poate conduce la erori de compatibilitate la rulare.

Sa urmarim 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 ca a disparut și cast-ul. De această dată, verificarea tipurilor este efectuată de compilator, ceea ce elimină potențialele erori de execuție cauzate de eventuale cast-uri incorecte. La modul general, beneficiile dobândite prin utilizarea genericității constau in:

* îmbunătățirea lizibilității codului
* creșterea gradului de robustețe

Metodele generice in Java preiau un parametru si returneaza un anumit tip de valoarea dupa executarea unui task. Functioneaza exact ca o metoda normala, doar ca metoda este declarata o singura data pentru un parametru care va fi de tip formal, apoi poate fi apelata cu parametri efectivi de orice tip.

class Test **{**

// A Generic method example

static **<**T**>** void genericDisplay**(**T element**)**

**{**

System**.**out**.**println**(**element**.**getClass**().**getName**()**

**+** " = " **+** element**);**

**}**

// Driver method

public static void main**(**String**[]** args**)**

**{**

// Calling generic method with Integer argument

genericDisplay**(**11**);**

// Calling generic method with String argument

genericDisplay**(**"GeeksForGeeks"**);**

// Calling generic method with double argument

genericDisplay**(**1.0**);**

**}**

**}**

Clasele generice sunt implementate la fel ca orice clasa normala in Java, cu diferenta ca va contine o sectiune ce contine tipuri formale de parametri (pot exista unul sau mai multe tipuri diferite).

class Test**<**T**>** **{**

T obj**;**

Test**(**T obj**)** **{** **this.**obj **=** obj**;** **}** // constructor

public T getObject**()** **{** **return** **this.**obj**;** **}**

**}**

class Main **{**

public static void main**(**String**[]** args**)**

**{**

Test**<**Integer**>** iObj **=** **new** Test**<**Integer**>(**15**);**

System**.**out**.**println**(**iObj**.**getObject**());**

Test**<**String**>** sObj **=** **new** Test**<**String**>(**"GeeksForGeeks"**);**

System**.**out**.**println**(**sObj**.**getObject**());**

**}**

**}**

class Test**<**T**,** U**>**

**{**

T obj1**;** // An object of type T

U obj2**;** // An object of type U

// constructor

Test**(**T obj1**,** U obj2**)**

**{**

**this.**obj1 **=** obj1**;**

**this.**obj2 **=** obj2**;**

**}**

// To print objects of T and U

public void print**()**

**{**

System**.**out**.**println**(**obj1**);**

System**.**out**.**println**(**obj2**);**

**}**

**}**

// Driver class to test above

class Main

**{**

public static void main **(**String**[]** args**)**

**{**

Test **<**String**,** Integer**>** obj **=**

**new** Test**<**String**,** Integer**>(**"GfG"**,** 15**);**

obj**.**print**();**

**}**

**}**

***Definirea unor structuri generice simple***

Să urmărim câteva elemente din definiția oferită 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 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 efectiv o structură generică, ele vor fi înlocuite cu tipurile efective utilizate în învocare. Concret, fie un apel de forma:

ArrayList**<**Integer**>** myList **=** **new** ArrayList**<**Integer**>();**

Iterator**<**Integer**>** it **=** **(**Iterator**<**Integer**>)** myList**.**iterator**();**

În această situație, tipul formal E a fost înlocuit (la compilare) cu tipul efectiv Integer. Observație: O analogie (simplistă) referitoare la acest mecanism de lucru cu tipurile se poate face cu mecanismul funcțiilor: acestea se definesc utilizând parametri formali, urmând ca, în momentul unui apel, acești parametri să fie înlocuiți cu parametri actuali.

***Genericitatea in subtipuri***

Să considerăm urmatoarea situație:

List**<**String**>** stringList **=** **new** ArrayList**<**String**>();** // 1

List**<**String**>** objectList **=** stringList**;**

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 operatie ar fi ilegala

Din acest motiv, operația 2 nu va fi permisă de către compilator! Generalizând, daca 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>.

***Wildcards***

Semnul întrebării <?> este cunoscut ca wildcard în programarea generică. Reprezintă un tip de date necunoscut. Wildcard-ul poate fi utilizat într-o varietate de situații, cum ar fi tipul unui parametru, câmp sau variabilă locală; uneori ca tip de returnare. Spre deosebire de array-uri, diferitele instanțieri de tip generic nu sunt compatibile între ele, nici măcar în mod explicit. Această incompatibilitate poate fi rezolvata de wildcard dacă <?> este folosit ca tip de parametru. Prin urmare, wildcard-urile sunt utilizate atunci când dorim să intrebuințăm 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 (ceea ce nu poate fi convertită la o colecție de un alt tip, după cum am vazut 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 insă este că nu putem adauga elemente arbitrare într-o colecție cu wildcard-uri:

Collection**<?>** c **=** **new** ArrayList**<**String**>();** // Operatie permisa

c**.**add**(new** Object**());**

Eroarea apare deoarece nu putem adăuga intr-o colecție generica decât elemente de un anumit tip, iar wildcard-ul nu indică un tip anume. Observație: 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 restricții asupra tipurilor ce pot inlocui un wildcard, obligându-le să se afle într-o relație ierarhica (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 afisa: "HamPizza", "CheesePizza", "Pizza"

**}**

**}**

Sintaxa List<? extends Pizza> 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, utilizand sintaxa List<? super Pizza>. Observație: Trebuie reținut faptul că în continuare nu putem introduce valori într-o colecție ce folosește bounded wildcards și este dată ca parametru unei funcții.

***Metode generice***

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

// 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 că correctCopy() este o metodă validă, care se execută corect, însă incorrectCopy() nu este, din cauza limitării pe care o cunoastem 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, 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**);**

***Aplicatii:***

1. Implementați o coadă de priorități care acceptă doar tipuri comparabile (descendente din java.lang.Comparable). Folosiți coada pentru a sorta în ordine crescătoare o serie de valori (veți testa valori numerice, generate aleator). Utilizați bounded wildcards.

* Aveți grijă la tipul parametrului cozii pe care o implementați (este restricționat)
* Pentru păstrarea la orice moment a ordinii valorilor conținute în colecția voastră, puteți să implementați coada ca o listă simplu-înlănțuită, în care capul listei să fie cel mai “mic” element
* Vă puteți crea o clasă internă care să încapsuleze un nod de listă înlănțuită
* Atenție la adăugarea de noduri în coadă și la cazurile extreme
* Folosiți o colecție generică în care veți stoca valorile numerice inițiale (de adăugat)
* Implementați o metodă generică de copiere a valorilor din această colecție în coada de priorități.
* Implementați colecția voastră ca iterabilă, compatibilă cu syntactic-sugar-ul for-each
* Trebuie să implementați interfața Iterable; atenție, și ea este generică
* Creați-vă iteratorul (parametrizat!) ca o clasă internă care să rețină datele necesare (nodul din listă la care a ajuns, etc.)
* nu este necesar să implementați metoda remove din Iterator
* Afișați-vă acum rezultatele folosind for-each pe coada voastră!

1. 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?