Programarea orientată pe obiecte

SERIA CB

GENERICITATEA

Genericitatea reprezintă capacitatea de a parametriza tipuri de date. Se pot defini *clase, interfețe sau metode cu tipuri generice*, pe care compilatorul le poate înlocui cu tipuri concrete. Cu ajutorul genericității, erorile pot fi detectate la compilare și nu la rulare. Compilatorul detectează erorile tipurilor de obiecte incompatibile.

Începând cu JDK 1.5, interfața `Comparable` are următoarea formă:

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o);
}
```

unde `<T>` reprezintă un tip formal generic ce poate fi înlocuit apoi cu un tip actual concret. Înlocuirea unui tip generic se numește instanțiere generică. `ArrayList` este o clasă generică începând cu JDK 1.5.

De exemplu, pentru a crea o listă de `String`-uri se folosește:

```
ArrayList<String> lista = new ArrayList<String>();
```

În acest fel, se pot adăuga doar obiecte de tipul `String` în listă. Dacă se încearcă adăugarea unui alt tip, va rezulta o eroare de compilare. Într-un `ArrayList` de tip generic nu pot fi adăugate tipuri primitive.

Următoarea sintaxă este greșită:

```
ArrayList<double> lista = new ArrayList<double> ();
```

Crearea corectă a unei liste cu elemente de tipul `Double` este:

```
ArrayList<Double> lista = new ArrayList<Double> ();
lista.add(5.5);
lista.add(3.0);
Double d1 = lista.get(0); // nu este nevoie de cast
double d2 = lista.get(1); // automat convertit la double
```

O clasă generică poate avea mai mult de un parametru. În acest caz, aceștia trebuie definiți astfel: `<T1, T2, T3>`.

METODE GENERICE

```
public class MetodaGenerica {
    public static void main(String[] args ) {
        Integer[] lista1 = {1, 2, 3, 4, 5};
        String[] lista2 = {"Hello", "World"};
        MetodaGenerica.<Integer>afisare(lista1);
        MetodaGenerica.<String>afisare(lista2);
    }

    public static<T> void afisare(T[] list) {
        for (int i = 0; i < lista.length; i++)
            System.out.print(lista[i] + " ");
        System.out.println();
    }
}
```

Un tip generic poate fi specificat ca subtip al altui tip. Acest tip generic este numit bounded.

```
<T extends numeClasa>
```

Un tip unbounded `<T>` are forma `<T extends Object>`.

Se poate crea o coadă generică astfel:

```
CoadaGenerica q = new CoadaGenerica();
```

Care este echivalent cu:

```
CoadaGenerica<Object> q = new CoadaGenerica<Object>();
```

O metodă ce returnează maximul a două elemente de tip generic este următoarea:

```
public class Max {
    public static <T extends Comparable<T>> T max(T o1 , T o2 ) {
        if( o1.compareTo(o2)> 0)
            return o1;
        else
            return o2;
    }
}
```

Se poate pasa o colecție generică ca argument al unei metode ce primește o colecție negenerică, însă pot apărea probleme. De exemplu, dacă se pasează `List<String>` într-o metodă declarată:

```
void metoda(List list) {
    list.add(valoare);
}
```

se primește un warning deoarece metoda poate fi considerată unsafe.

Atribuirile polimorfice se aplică doar la tipul de bază, nu și la parametrul generic. Este corectă atribuirea:

```
List<Animal> list = new ArrayList<Animal>();
```

Dar nu și atribuirea:

```
List<Animal> list = new ArrayList<Pisica>();
```

Această regulă se aplică peste tot unde este implicat polimorfismul:

```
void metoda(List<Animal> list){} // nu poate lua List<Pisica>
List<Animal> metoda2() {} // nu poate returna List<Pisica>
```

WILDCARDS

Fie următorul program:

```
public class WildCard {
    public static void main(String[] args ) {
        Stiva<Integer> s = new Stiva<Integer>();
        s.push(1);
        s.push(2);
        s.push(3);
        System.out.print(max(s));
    }

    public static double max(Stiva<Number> st) {
        double max = st.pop().doubleValue();
        while (!st.isEmpty()) {
            double val = st.pop().doubleValue();
            if (val > max)
                max = val;
            }
        return max;
    }
}
```

În exemplul de mai sus, nu putem apela `max(s)` deoarece `Stiva<Integer>` nu este un subtip al lui `Stiva<Number>`, chiar dacă `Integer` este un subtip al lui `Number`. O definire corectă a metodei max este:

```
public static double max(Stiva<? extends Number> st)
```

Unde `<? extends Number>` este un tip wildcard ce reprezintă `Number` sau un subtip al lui `Number`.

<?> este un wildcard ce reprezintă orice obiect și este echivalent cu <? extends Object>.

Un exemplu care folosește wildcard-ul `<? super T>` este:

```
public class WildCard {
    public static void main(String[] args) {
        List<Double> lista1 = new ArrayList<Double>();
        List<Object> lista2 = new ArrayList<Object>();
        lista1.add(1.25);
        lista1.add(2.25);
        lista2.add("Ana");
        reuniune(lista1, lista2);


    }

    1 usage
    public static <T> void reuniune(List <T> lista1,  List<? super T> lista2) {
        for(int i = 0; i < lista1.size(); i++)
            lista2.add(lista1.get(i));
        System.out.println(lista2);
    }
}
```

Cuvântul cheie `extends` poate însemna `extends` sau `implements`. În `<? extends Animal>`, `Animal` poate fi o clasă sau o interfață.

Dacă `lista1` și `lista2` sunt create după cum urmează:

```
ArrayList<String> lista1 = new ArrayList<String>();
ArrayList<Integer> lista2 = new ArrayList<Integer>();
```

Apelul:

```
T object = new T();
```

Este greșit. `T` este un tip generic care este șters la runtime.

Nici apelul:

```
T[] elemente = new T[dimensiune];
```

nu este corect.

La fel, apelul:

```
ArrayList<String>[] lista = new ArrayList<String>[10];
```

este incorect.

Versiunea corectă este:

```
ArrayList<String>[] lista = (ArrayList<String>[]) new ArrayList[10];
```

Nu ne putem referi la tipuri generice într-un context static.

Declarația:

```
public static T o1;
```

nu este permisă.

O clasă generică nu poate extinde `java.lang.Throwable`. Următoarea declarație nu este permisă:

```
public class Exceptie<T> extends Exception
```

În acest caz, JVM trebuie să verifice dacă excepția este compatibilă cu tipul specificat în clauza `catch`. Acest lucru este imposibil, deoarece tipul informației nu este prezent la runtime.

```
public class Exceptie<T> extends Exceptiontry {
...
}
catch (Exceptie<T> ex) {
...
}
```

Sintaxa wildcard-urilor se aplică la metode generice și acceptă subtipuri sau supertipuri ale tipului declarat ca argument al metodei.

```
void adauga(List<Animal> lista) {} // poate lua doar <Animal>
void adauga(List<? extends Animal>) {} // poate lua <Animal> sau <Pisica>
```

```
List<? extends Number> myList = new ArrayList<Integer>();
myList.add(new Integer(3));
```

Acest exemplu nu va compila, deoarece la compilare Java nu știe ce tip este `List<? extends Number>`. Astfel, la compilare, `myList` trebuie să fie `List<Double>` sau `List<Integer>` sau un `List` ce conține elemente subclasă a lui `Number`.

Fie următorul exemplu:

Sintaxa `List<? extends Automobil>` presupune ca tipul elementelor listei să fie `Automobil` sau o subclasă a acesteia. Astfel, variabila `lista` poate avea tipul `List<Dacia>` sau `List<BMW>`. Pentru ca tipul elementelor să fie `Automobil` sau o superclasă a acesteia, se poate utiliza sintaxa `List<? super Automobil>`.

```java
class Automobil {

    protected String marca = "Automobil";

    public String getMarca() {
        return marca;
    }
}

class Dacia extends Automobil {
    public Dacia() {
        marca = "Dacia";
    }
}

class BMW extends Automobil {
    public BMW() {
        marca = "BMW";
    }
}

public class Test {
    public static void listAutomobil(List<? extends Automobil> lista) {
        for(Automobil a : lista)
            System.out.println(a.getMarca());
    }

    public static void main(String[] args) {
        List<Automobil> lista = new ArrayList<Automobil>();
        lista.add(new Dacia());
        lista.add(new BMW());
        lista.add(new Automobil());
        listAutomobil(lista);
    }
}
```

APLICAȚII:

Lista

Implementați clasa generică `Lista<E>` ce conține o listă cu ajutorul unui tablou de elemente de tipul `E`. Ea conține variabila membru `private tablou`, de tipul `E[]` și variabila membru `private int nr`, ce reprezintă numărul curent de elemente din listă. Să se definească următoarele metode:

- `public Lista(int dim)` – constructor cu parametru ce inițializează lista cu capacitatea `dim`. Dacă `dim` este mai mic sau egal cu `0`, se va arunca excepția `IllegalArgumentException`. Aici se vor inițializa variabilele `tablou` si `nr`.
  `tablou = (E[]) new Object[dim];`
- `public void adauga(E x)` – adaugă elementul `x` în listă. Dacă s-a depășit în momentul actual capacitatea listei, se va redimensiona la capacitate dublă prin copierea element cu element a valorilor într-un vector auxiliar ce va fi apoi atribuit referinței tablou. Se poate utiliza și o metodă ajutătoare numită `public void redimensioneaza()`.
- `public void afiseaza()` – afișează elementele listei
- `public boolean cauta(E x)` – caută elementul `x` în listă și returnează valoarea booleană corespunzătoare. Pentru a compara elementele, se poate utiliza metoda `equals()`.

În programul principal, să se creeze `3` liste – cu elemente de tipul `Integer / Double / String`, și apoi să se apeleze metodele definite mai sus pentru fiecare dintre ele. Să se afișeze conținutul fiecărei liste și să se trateze și situațiile de excepție.

Exemplu:

```
Lista<Integer> lista1 = new Lista<>(3);
```

Magazin

Implementați clasa abstractă `Produs`, ce implementează interfața `Comparable`. Ea conține metodele:

- `double pretRaft()` – returnează prețul la raft al produsului – metodă abstractă;
- `void afiseaza()` – afișează informațiile despre produs – metodă abstractă;
- `public int compareTo (Object p)` – suprascrie metoda `compareTo` din interfața `Comparable`, astfel încât metoda returnează `1` dacă prețul la raft al produsului curent este mai mare decât prețul la raft al produsului `p`, `-1`, dacă prețul la raft al produsului curent este mai mic decât prețul la raft al produsului `p` și `0` daca au prețurile la raft egale.

Implementați clasele `ProdusAlimentar`, `ProdusCuratenie`, `ProdusIgiena` (cu variabila membru `double pretProducator`).

Fiecare dintre aceste clase extind `Produs`, conțin un constructor cu parametri și suprascriu metodele `pretRaft()` și `afiseaza()`. Pentru produsul alimentar se aplică un adaos comercial de `20%`, pentru cel de curățenie de `15%` iar pentru cel de igienă de `10%`. Astfel, de exemplu, prețul la raft al unui produs alimentar este cu `20%` mai mare decât cel furnizat de producător.

Să se definească o metodă care determină și afișează informațiile despre obiectul cu prețul la raft maxim dintr-o listă cu obiecte de tipul `Produs`. Aceasta are prototipul:

```
public static void pretRaftMaxim(ArrayList<? extends Produs> lista)
```

În programul principal, să se creeze un `ArrayList` de obiecte de tipul `Produs`, să se introducă obiecte în el și apoi să se apeleze metoda `pretRaftMaxim` pentru a determina produsul cu prețul la raft maxim.

Referințe:

1. Oana Balan, Mihai Dascalu. **Programarea orientata pe obiecte in Java**. Editura Politehnica Press, 2020
2. Bert Bates, Kathy Sierra. **Head First Java: Your Brain on Java - A Learner's Guide 1st Edition**. O'Reilly Media. 2003
3. Herbert Schildt. **Java: A Beginner's Guide**. McGraw Hill. 2018
4. Barry A. Burd. **Java For Dummies**. For Dummies. 2015
5. Joshua Bloch. **Effective Java**. ISBN-13978-0134685991. 2017
6. Harry (Author), Chris James (Editor). **Thinking in Java: Advanced Features (Core Series) Updated To Java 8**
7. Learn Java online. Disponibil la adresa: https://www.learnjavaonline.org/
8. Java Tutorial. Disponibil la adresa: https://www.w3schools.com/java/

   The Java Tutorials. Disponibil la adresa: https://docs.oracle.com/javase/tutorial/