



# Tipuri **GENERIC** de date în **JAVA 5**

Claudiu Soroiu

**Java, ca orice limbaj de programare sau mai bine zis, aplicație a evoluat de-a lungul timpului. Deși până nu cu mult timp în urmă cea mai nouă versiune era 1.4.2 și se credea că va urma versiunea 1.5, s-a decis ca noua versiune să fie 5.0. Noua versiune oferă noi facilități programatorilor pentru a-i ajuta să realizeze aplicații mai stabile într-un timp mai scurt. Una dintre aceste facilități o reprezintă introducerea tipurilor generice. Această facilitate va fi prezentată în cadrul acestui articol.**

Vom începe mai întâi prin a aminti faptul că ultima versiune de *Java* este 5.0, deși identificatorul de versiune se prezintă sub forma 1.5.0.

Dacă în versiunea anterioară a platformei *Java* s-a dorit îmbunătățirea calității a setului standard de clase, de această dată platforma *Java* a suferit modificări la nivelul mașinii virtuale.

*Genericitatea* sau mai bine zis *tipurile parametrizate* reprezintă un mecanism cu ajutorul căruia programatorii pot specifica tipul obiectelor cu care o anumită clasă va opera prin intermediul parametrilor.

Pentru a înțelege mai bine acest concept, amintim că tipurile parametrizate de date sunt, într-o oarecare măsură similare *template*-urilor din C++.

Pentru a prezenta mai bine acest concept ne vom folosi de colecțiile de date.

În versiunile anterioare, colecțiile de date puteau conține orice obiect, existând oricând riscul apariției unei excepții la *runtime* de tipul *ClassCastException* datorită introducerii

mai mult sau mai puțin voluntare într-o colecție a unui obiect care are alt tip decât cel pentru care se folosea colecția respectivă.

În continuare vă vom prezenta un exemplu de program în care apare excepția amintită anterior:

```
import java.util.ArrayList;

public class Test1 {
    static ArrayList a =
        new ArrayList();
    public static void main
        (String[] args) {
        a.add("Gazeta");
        a.add("de");
        a.add(new Integer(4));
        a.add("Informatica");
        a.add("GInfo")
        print();
    }

    public static void print() {
        for (int i = 0; i < a.size();
            i++) {
            String s = (String)
                a.get(i);
            System.out.print(s +
                " ");
        }
    }
}
```

```
System.out.println();
}
```

Codul anterior se compilează corect, dar la execuție, în cadrul metodei `print()` este generată o excepție *ClassCastException* datorită faptului că un obiect de tipul *Integer* nu poate fi convertit la tipul *String*.

În *Java 5* tipul obiectelor care vor fi stocate într-o colecție poate fi precizat în momentul declarării colecției, astfel, codul următor poate fi transcris în *Java 5* în:

```
import java.util.ArrayList;

public class Test1 {
    static ArrayList<String> a =
        new ArrayList<String>();

    public static void main
        (String[] args) {
        a.add("Gazeta");
        a.add("de");
        a.add(new Integer(4));
        a.add("Informatica");
        print();
        a.add("GInfo")
    }
}
```



```
public static void print() {
    for (int i = 0; i < a.size(); i++) {
        String s = a.get(i);
        System.out.print(s + "
");
    }
    System.out.println();
}
```

Acest cod nu poate fi compilat deoarece s-a declarat o colecție care conține obiecte de tipul `String`, iar în cadrul metodei `main` se încearcă adăugarea unui obiect de tipul `Integer`.

În momentul eliminării unui obiect dintr-o colecție pentru care s-a specificat tipul obiectelor conținute, nu mai este necesară conversia rezultatului la tipul dorit.

În metoda `print`, linia de cod `a.get(i)` returnează un obiect de tipul `String`.

În continuare vom prezenta modul în care se pot declara tipuri parametrizate de date:

```
public class NV <N,V> {
    private N name;
    private V value;

    public NV(N name, V value) {
        this.name = name;
        this.value = value;
    }

    public N getName() {
        return name;
    }

    public V getValue() {
        return value;
    }

    public static void main
        (String[] args) {
        NV<String, Integer> nv =
            new NV<String, Integer>
                ("Ionel", new Integer(4));
        System.out.println(
            nv.getName());
        System.out.println(
            nv.getValue());
    }
}
```

După cum se poate observa în cadrul codului anterior, pentru a declara parametrii unei clase se folosesc caracterele "<" și ">". Între aceste caractere se află lista parametrilor.

O clasă parametrizată poate fi folosită ca și cum nu ar fi parametrizată, implicit valoarea parametrilor fiind `Object`.

Faptul că anumite clase sunt parametrizate este utilizat doar de compilatorul *Java* nu și de mașina virtuală, codul generat de compilatorul *Java 5* putând fi executat și de versiunea 1.4 a platformei.

Domeniul de definiție al parametrilor unei clase poate fi restrâns în sensul că poate fi specificat tipul acestora (clasele și/sau interfețele din care aceștia trebuie să fie derivați).

În continuare vă prezentăm un exemplu de clasă ai cărei parametri au fost restricționați:

```
public class NV <N,V extends
    Number> {

    private N name;
    private V value;

    public NV(N name, V value) {
        this.name = name;
        this.value = value;
    }

    public N getName() {
        return name;
    }

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

Un cod care ar conține o declarație de forma: `NV<String, Date>` nu ar putea fi compilat deoarece cel de-al doilea parametru al clasei trebuie să extindă clasa `Number`.

Domeniul de definiție al parametrilor poate fi restrâns astfel încât anumiți parametri să implementeze mai multe interfețe. O declarație în care un parametru implementează mai multe interfețe poate fi:

```
public class NV <N,V extends
    Number & Serializable>
```

Introducerea tipurilor parametrizate de date a dus la modificarea modului de declarație a metodelor, astfel că pot apărea secvențe de cod ca:

```
void print (NV<? extends X, Y>);
sau
void print (NV<? super X, Y>).
```

În cadrul primei declarații, metoda `print()` primește ca parametru un obiect de tipul `NV`, iar primul parametru al clasei `NV` trebuie să fie derivat din `X`, unde `X` este un parametru al clasei curente.

În cadrul primei declarații, metoda `print()` primește ca parametru un obiect de tipul `NV`, iar primul parametru al clasei `NV` trebuie să fie părintele lui `X`, unde `X` este un parametru al clasei curente.

Parametrii claselor pot fi utilizați și pentru a arunca excepții în *Java 5*.

```
public interface Except
    <E extends Exception> {
    public void doRun() throws E;
}

public class Action implements
    Except<FileNotFoundException> {
    public void doWork() throws
        FileNotFoundException {
        new File("/foo/bar.txt");
    }
}
```

O clasă poate să extindă o clasă parametrizată și/sau să suprascrie metode care se folosesc de parametrii clasei.

Regula de bază în momentul moștenirii unei clase sau suprascrierii unei metode este aceea că tipurile de date utilizate să fie subtipuri ale tipurilor definite în *super*-clasa sau într-o *super*-interfață.

Dacă înainte de *Java 5* metodele care suprascriau alte metode trebuiau să returneze același tip de date, de această dată trebuie să returneze un tip de dată derivat din tipul de dată al metodei suprascrise.

