

Објектно орјентисано програмирање



Владимир Филиповић

vladaf@matf.bg.ac.rs

Енумерисани и генерички ТИПОВИ



Владимир Филиповић

vladaf@matf.bg.ac.rs



Енумерисани тип

Понекад променљива служи за чување вредности из ограниченог скупа.

На пример, можда се продаје одећа у четири величине: `small`, `medium`, `large` и `extra large`. Наравно, ове величине се могу кодирати целим бројевима 1, 2, 3, 4 или знацима S, M, L, X. Али такав приступ је склон грешкама. Веома лако се може догодити да променљива добије погрешну вредност (попут 0 или `m`).

Почев од верзије 5.0 могуће је у Јави дефинисати сопствени енумерисани тип. Такав тип има коначан број именованих вредности.

На пример, за претходно описану ситуацију уводи се енумерисани тип:

```
enum Velicina {SMALL, MEDIUM, LARGE, EXTRA_LARGE};
```



Енумерисани тип (2)

Енумерисани тип се заправо понаша као класа.

Класа из претходног примера садржи тачно четири податка тј. тачно четири примерка - није могуће конструисати нове објекте.

Према томе, нема потребе користити метод `equals()` за поређење вредности енумерисаног типа, већ се подаци тог типа пореде коришћењем оператора `==`.

Могуће је декларисати променљиву енумерисаног типа. На Пример. `Velicina v = Velicina.MEDIUM;`

Променљива типа `Velicina` може чувати само једну од вредности из листаних у декларацији овог типа или специјалну вредност `null`.



Енумерисани тип (3)

Могуће је, по потреби, додати конструкторе, методе и поља типу енумерације. Наравно, конструктори се позивају само приликом конструисања константи енумерације. Следи Пример.

```
enum Velicina
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String skracenica;

    private Velicina(String skracenica)
    {
        this.skracenica = skracenica;
    }
    public String getSkracenica()
    {
        return skracenica;
    }
}
```



Енумерисани тип (4)

Сви типови енумерације су поткласе класе Enum. Они наслеђују извештајан број метода од те класе.

Најкориснији међу њима је метод toString(), који враћа име константе енумерације. Тако, на пример, Velicina.SMALL.toString() враћа стринг „SMALL“.

Супротан ефекат има статички метод valueOf(). На пример наредба,

```
Velicina v = (Velicina) Enum.valueOf(Velicina.class, "SMALL");
```

поставља v на Velicina.SMALL.



Енумерисани тип (5)

Сваки тип енумерације поседује статички метод `values()` који враћа низ вредности енумерације.

Тако, на пример, наредба

```
Velicina[] vrednosti = Velicina.values();
```

враћа низ са елементима

`Velicina.SMALL`, `Velicina.MEDIUM`, `Velicina.LARGE` и `Velicina.EXTRA_LARGE`.

Метод `ordinal()` враћа позицију константе енумерације у `enum` декларацији, при чему бројање почиње од 0.

Тако, на пример, `Velicina.MEDIUM.ordinal()` враћа 1.



Енумерисани тип (6)

Енумерације се могу користити код вишеструког гранања помоћу наредбе `switch`:

```
Velicina v = . . . ;  
switch (v)  
{  
    case SMALL: // nema potrebe koristiti Velicina.SMALL  
        . . .  
        break;  
    . . .  
}
```

У `case` клаузама не наводи се тип енумерације, већ само име константе.



Енумерисани тип (7)

java.lang.Enum<E>

- `static Enum valueOf(Class enumClass, String name)`
враће енумерисану константу датог типа која има дато име.
- `String toString()`
враће име енумерисане константе.
- `int ordinal()`
враће позицију енумерисане константе у декларацији `enum`, при чему одбројавање почиње од нуле.
- `int compareTo(E other)`
враће негативан цео број ако је дата енумерисана константа пре константе `other` у набрајању, враће нулу акоа је `this == other`, а иначе враће позитивни цео број.



Појам генеричког типа

Генерички тип омогућује да при дефинисању класа, интерфејса и метода сами типови (тј. класе и интерфејси) буду параметри.

Дакле, генерички типови се понашају слично као формални параметри при дефинисању метода. На тај начин се постиже поновна искористивост истог кода за различите улазе.

Програмирање коришћењем генеричких типова има следеће предности:

- Строжија контрола типа приликом превођења Јава програма.
- Елиминација експлицитне конверзије типа (кастовања).
- Омогућавање да се имплементирају генерички алгоритми.



Предности генеричког типа

```
List stones = new LinkedList();  
stones.add(new Stone(RED));  
stones.add(new Stone(GREEN));  
stones.add(new Stone(RED));  
Stone first = (Stone) stones.get(0);
```

Код старог приступа
експлицитна конверзија
је досадна али
суштински неопходна!

```
public int countStones(Color color) {  
    int tally = 0;  
    Iterator it = stones.iterator();  
    while (it.hasNext()) {  
        Stone stone = (Stone) it.next();  
        if (stone.getColor() == color) {  
            tally++;  
        }  
    }  
    return tally;  
}
```



Предности генеричког типа (2)

У новом приступу
интерфејс List је
генерички интерфејс
коме тип елемента
листе представља
параметар



```
List<Stone> stones = new LinkedList<Stone>();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = /*no cast*/ stones.get(0);
```

```
public int countStones(Color color) {
    int tally = 0;
    /*no temporary*/
    for (Stone stone : stones) {
        /*no temporary, no cast*/
        if (stone.getColor() == color) {
            tally++;
        }
    }
    return tally;
}
```



Предности генеричког типа (3)

стари
начин

```
List stones = new LinkedList();  
stones.add("ovo nije kamen");  
  
...
```

```
Stone stone = (Stone) stones.get(0);
```

Нема провере

Грешка при извршавању
Runtime error

НОВИ
начин

```
List<Stone> stones = new LinkedList<Stone>();  
stones.add("ovo nije kamen");  
  
...
```

```
Stone stone = stones.get(0);
```

Провера при
превођењу

Извршење је без
проблема



Стек као генерички интерфејс

```
public interface StackInterface {  
    public boolean isEmpty();  
    public int size();  
    public void push(Object item);  
    public Object top();  
    public void pop();  
}
```

Стари начин

```
public interface StackInterface<E> {  
    public boolean isEmpty();  
    public int size();  
    public void push(E item);  
    public E top();  
    public void pop();  
}
```

Нови начин:
Дефинише се
генерички
интерфејс
који има **тип**
као
параметар



Дефинисање генеричког типа

Генерички тип је у ствари генеричка класа или интерфејс са параметризованим типовима тј. чија дефиниција садржи параметре који представљају типове.

Пример.

Следећа класа, названа `Box`, ће бити модификована тако да опише концепт генеричког типа. На почетку се ради о једној обичној класи.

```
public class Box
{
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```



Дефинисање генеричког типа (2)

Генеричка класа се дефинише на следећи начин:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

Секција са параметрима који представљају типове, ограничена са знацима < и > следи непосредно иза имена класе. У тој секцији се специфицирају параметри који представљају типове T1, T2, ..., и Tn.

Сада класа Box има следећу структуру:

```
/**  
 * Generic version of the Box class.  
 * @param <T> the type of the value being boxed  
 */  
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```



Дефинисање генеричког типа (3)

Конвенције за именовање параметара који представљају тип код генеричких типова, интерфејса и метода:

По конвенцији, параметри су означени једним великим словом, што се на први поглед разликује од имена поља, парамеатар и метода.

Најчешће се користе следеће ознаке:

- Е - Елеменат (енг. Element - њега ексклузивно користи Java Collections Framework)
- К – Кључ (енг. Key)
- N – Број (енг. Number)
- Т – Тип (енг. Type)
- V – Вредност (енг. Value)
- S,U,V итд. – други, трећи, четврти итд. тип



Генерички позив типа

Да би се из програмског кода реферисало на генеричку класу `Box`, потребно је извршити **генерички позив типа**, којим се параметар `T` мења са неком конкретном вредношћу, као што је `Integer`:

```
Box<Integer> integerBox;
```

Генерички позив типа се може посматрати слично обичном позиву метода, али уместо што се вредност аргумента прослеђује методу у овом случају се прослеђује аргумент који представља тип (у овом случају тип `Integer`) самој класи (у овом случају класи `Box`).

Позивање генеричког типа се обично означава као **параметарски тип**.

За креирање примерка класе користи се кључна реч `new`, али се између имена класе и заграда поставља `<Integer>`:

```
Box<Integer> integerBox = new Box<Integer>();
```



Генерички позив типа (2)

Програмски језик Јава допушта, почев од верзије 7, да се у конструктору примерка за генеричку класу, ознака типа код оператора `new` замени са паром знакова `<>`, док год преводилац може на основу контекста да одреди ког типа треба да буду аргументи типови. Пар знакова `<>` се у жаргону назива дијамант.

For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

Аргумент који представља тип се може приликом генеричког позива заменити са конкретним типом (нпр. са `Integer` или са `Student`) али се може заменити и са параметризованим типом (нпр. са `Box<Double>` или са `List<String>`).



Генерички метод

Генерички методи су они методи са параметрима који представљају типове. Они су слични генеричким типовима, али је опсег параметара који представљају типове ограничен на метод у ком су ти параметри декларисани.

Синтакса за генерички метод садржи (пре повратног типа) параметре који представљају тип између знака $<$ и $>$, а ти параметри се пишу пре типа повратне вредности у декларацији метода.

Пример. Класа Util садржи генерички метод compare за поређење два примерка генеричке класе Pair:

```
public class Util {  
    // Generic static method  
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
        return p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue());  
    }  
}
```



Генерички метод (2)

Пример (наставак): Сама класа Pair има следећу структуру:

```
public class Pair<K, V>
{
    private K key;
    private V value;

    // Generic constructor
    public Pair(K key, V value)
    {
        this.key = key;
        this.value = value;
    }

    // Generic methods
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey() { return key; }
    public V getValue() { return value; }
}
```



Генерички метод (3)

Пример (наставак): Синтакса за позив метода који пореди примерке класе `Pair` је следећа:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.<Integer, String>compare(p1, p2);
```



Ограничења за типове

Понекад је потребно да генеричка класа или генерички метод постави ограничење на променљиве типа.

Пример: Треба одредити најмањи елеменат низа:

```
class ArrayAlg
{
    public static <T> T min(T[] a) // almost correct
    {
        if (a == null || a.length == 0) return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
            if (smallest.compareTo(a[i]) > 0) smallest = a[i];
        return smallest;
    }
}
```

Променљива `smallest` је типа `T`, што значи да она може бити примерак ма које класе. Одмах се поставља питање: како можемо знати да класа `T` садржи метод `compareTo`?



Ограничења за типове (2)

Решење је да се ограничи параметар типа `T` само на оне класе које имплементирају интерфејс `Comparable` — стандардни интерфејс који садржи само један метод, `compareTo`. На тај начин ће бити обезбеђено да позив метода `compareTo` не доведе до грешке.

Ово се постиже постављањем ограничења на променљиву типа `T`:

```
public static <T extends Comparable> T min(T[] a) . . .
```

У овом случају, интерфејс `Comparable` је и сам генерички тип. Сада се генерички метод `min` може позивати само над низовима примерака класа које имплементирају интерфејс `Comparable`, као што су `String`, `Date` итд. Позив овог метода над низом објеката неке класе која не имплементира интерфејс `Comparable` доводи до грешке у превођењу.



Ограничења за типове (3)

Поставља се питање: зашто се у претходном случају користи кључна реч `extends`, а не нпр. `implements` – на крају крајева, `Comparable` је интерфејс, а не класа.

Нотација

`<T extends TipKojiOgranicava>`

изражава да параметарски тип `T` треба да буде подтип типа који ограничава. При томе, и параметарски тип `T` и тип који ограничава могу бити и класе и интерфејси.

Кључна реч `extends` најбоље описује такав однос међу њима, тј. концепт подтипова - узевши у обзир да дизајнери Јаве нису желели да додају нову кључну реч (нпр. `sub`) у језик.



Ограничења за типове (4)

Параметар типа може имати више ограничења. На пример:

`T extends Comparable & Serializable`

Типови који ограничавају су одвојени знаком `&` зато што се знак зарез користи за раздвајање променљивих типа.

Исто као и код Јава наслеђивања, могуће је имати више интерфејса који ограничавају параметар типа, али само једно ограничење може да се односи на класу. Ако класа представља ограничење са тип, назив класе се мора навести пре назива интерфејса.

Пример. Метод `minmax`, који одређује минимум и максимум низа, је направљен тако да буде генерички:

```
import java.util.*;  
public class PairTest2  
{
```



Ограничења за типове (5)

Пример (наставак).

```
public static void main(String[] args)
{
    GregorianCalendar[] birthdays =
    {
        new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
        new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelac
        new GregorianCalendar(1903, Calendar.DECEMBER, 3), // v. Neumann
        new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
    };
    Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
    System.out.println("min = " + mm.getFirst().getTime());
    System.out.println("max = " + mm.getSecond().getTime());
}
}
```



Ограничења за типове (6)

Пример (наставак).

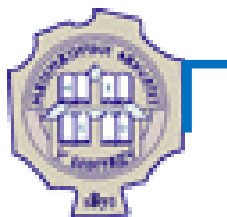
```
class ArrayAlg
{
    public static <T extends Comparable> Pair<T> minmax(T[] a)
    {
        if (a == null || a.length == 0) return null;
        T min = a[0];
        T max = a[0];
        for (int i = 1; i < a.length; i++)
        {
            if (min.compareTo(a[i]) > 0) min = a[i];
            if (max.compareTo(a[i]) < 0) max = a[i];
        }
        return new Pair<T>(min, max);
    }
}
```



Генерици и виртуелна машина

Кад год се дефинише генерички тип, аутоматски бива обезбеђен одговоарајући сирови (енг. raw) тип.

Име сировог типа је исто као име генеричког типа, само што су уколоњени параметри који представљају типове. Променљиве које представљају типове су просто замењене са типовима који их ограничавају или са Object типом (ако за те променљиве није било ограничења).



Генерици и виртуелна машина (2)

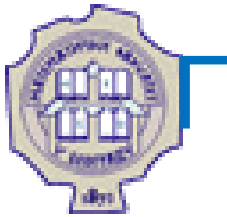
Пример.

Сирови тип за `Pair<T>` има следећи облик:

```
public class Pair
{
    private Object first;
    private Object second;

    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }
}
```



Генерици и виртуелна машина (3)

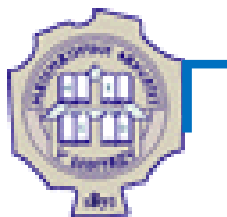
У претходном примеру параметар T није био ограничен, па је стога он једноставно замењен са класом tj . типом `Object`.

Мада програм може садржати различите врсте парова, као што су `Pair<String>` или `Pair<GregorianCalendar>`, током превођења сви они буду преведени у сирове `Pair` типове.

Пример.

Претпоставимо да је донекле другачије дефинисан тип:

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
    private T lower;
    private T upper;
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
    ...
}
```



Генерици и виртуелна машина (4)

Пример (наставак).

Тада сирови `Interval` тип има следећи облик:

```
public class Interval implements Serializable
{
    private Comparable lower;
    private Comparable upper;
    public Interval(Comparable first, Comparable second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
    ...
}
```

Ако се промени редослед граница у дефиницији класе `Interval` :

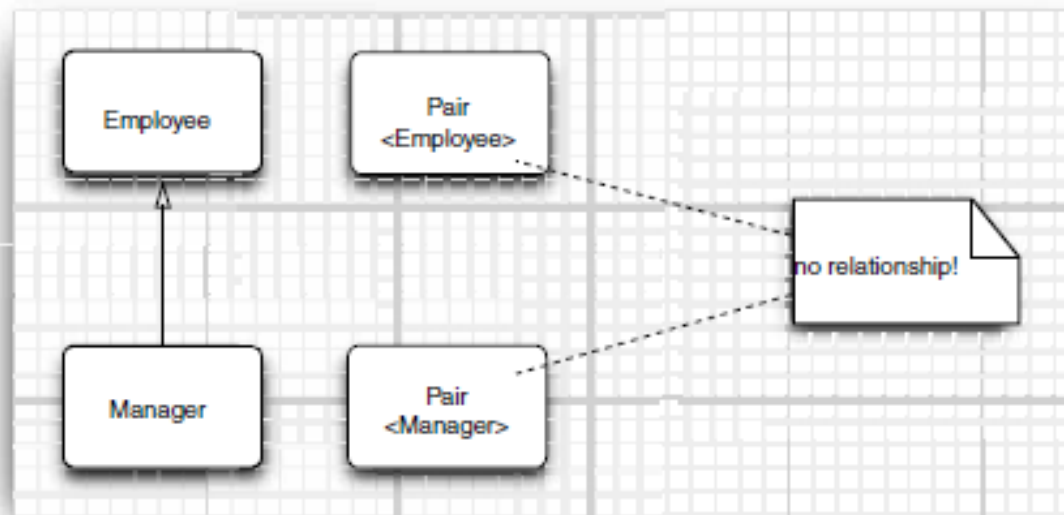
```
class Interval<Serializable & Comparable>
```

тада се код сировог типа `T` замењује са `Serializable` и преводилац убацује конверзије у `Comparable` где год је то неопходно.



Генерици и наслеђивање

Уопштено говорећи, не постоји веза између $\text{Pair}<S>$ и $\text{Pair}<T>$, без обзира на то како су S и T повезани.



У Јави је класа `Manager` подкласа класе `Employee` само ако интерфејси поткласе укључују све интерфејсе надкласе. Међутим, направљени генерици обично имају различите интерфејсе па не може бити наслеђивања.



Генерици и наслеђивање (2)

У примеру који следи, преводилац спречава превођење кода са операцијама које нису безбедне за тип – није могуће са листом ниски радити као са листом објеката, без обзира на то што је класа `String` изведена из класе `Object`.

```
List<String> ls = new ArrayList<String>();  
  
List<Object> lo = ls;  
  
lo.add(0, new Object()); // legal?!  
ls.get(0); // Not a string?!
```

Грешка при
превођењу јер
конверзија није
безбедна за тип!



Закључак

Као закључак, потребно је водити рачуна о следећим чињеницама када се ради са Јава генерицима:

- У раду Јава виртуелне машине не постоје генерици, већ само обичне класе и методи.
- Сви типовни параматри бивају замењени са својим границама.
- Преводацац генерише методе-мостове и на тај начин задржава полиморфно понашање.
- Да би се обезбедила безбедност типова, преводацац тамо где је потребно убацује наредбе за експлицитну конверзију типова.



Захвалница

Велики део материјала који је укључен у ову презентацију је преузет из презентације коју је раније (у време када је он држао курс Објектно орјентисано програмирање) направио проф. др Душан Тошић.

Хвала проф. Тошићу што се сагласио са укључивањем тог материјала у садашњу презентацију, као и на помоћи коју ми је пружио током конципирања и реализације курса.

Надаље, један део материјала је преузет од колегинице Марије Милановић.

Хвала Марији Милановић на помоћи у реализацији ове презентације.