

Java tečaj

Tehnologija Java Generics

- Kako se parametrizacija koju nudi tehnologija Java Generics obilno koristi u Javinom okviru kolekcija, najprije ćemo se upoznati s ovom tehnologijom, i to kroz primjer
- Prisjetimo se razreda `Integer`, `Double`, `Boolean` i sličnih
 - To su omotači (engl. `Wrappers`) oko primitivnih vrijednosti a kako nude samo gettere, vrijednosti su nepromjenjive
- Pretpostavimo da trebamo sličnu funkcionalnost:
 - Metodi želimo predati broj koji metoda po potrebi može izmijeniti (ali tako da je po povratku iz metode ta promjena vidljiva u glavnom programu)

■ Je li ovo rješenje?

```
public class Main {  
  
    public static void main(String[] args) {  
        Integer number = new Integer(5);  
        System.out.println("Before: " + number);  
        updateRandomly(number);  
        System.out.println("After: " + number);  
    }  
  
    private static void updateRandomly(Integer number) {  
        if(Math.random() < 0.5) {  
            number = number + 1;  
        }  
    }  
}
```

- Napišimo razred koji će “omotati” objekt i pošaljimo omotač metodi
 - Potrebna disciplina: objekt moramo dohvaćati pozivom gettera

```
public class IntWrapper {  
    private Integer value;  
  
    public IntWrapper() { this.value = Integer.valueOf(0); }  
    public IntWrapper(Integer value) { this.value = value; }  
    public Integer getValue() { return value; }  
    public void setValue(Integer value) { this.value = value; }  
  
    @Override  
    public String toString() {  
        if(value==null) return "";  
        return value.toString();  
    }  
}
```

- Napišimo razred koji će “omotati” objekt i pošaljimo omotač metodi
 - Potrebna disciplina: objekt moramo dohvaćati pozivom gettera

```
public class Main {  
  
    public static void main(String[] args) {  
        IntWrapper number = new IntWrapper(5);  
        System.out.println("Before: " + number);  
        updateRandomly(number);  
        System.out.println("After: " + number);  
    }  
  
    private static void updateRandomly(IntWrapper number) {  
        if(Math.random() < 0.5) {  
            number.setValue(number.getValue()+1);  
        }  
    }  
}
```

Autoboxing / autodeboxing

- U kôdu se mogu javiti situacije u kojima se na nekom mjestu očekuje wrapper a predana je primitivna vrijednost
 - Prevoditelj će sam umetnuti kôd koji će primitivnu vrijednost zamotati u odgovarajući objekt:

```
IntWrapper number = new IntWrapper(5); // očekuje se Integer  
                                         // kao argument
```

```
// Prevoditelj će kod preoblikovati u (autoboxing):
```

```
IntWrapper number = new IntWrapper(Integer.valueOf(5));
```

Autoboxing / autodeboxing

- U kôdu se mogu javiti situacije u kojima se na nekom mjestu očekuje primitivna vrijednost a predan je wrapper
 - Prevoditelj će sam umetnuti kôd koji će iz wrappera dohvatiti primitivnu vrijednost:

```
number.setValue(number.getValue()+1); // Uočite zbrajanje: Integer + int
// Kako bi zbrajanje bilo legalno, prevoditelj će dodati poziv (deboxing):
number.setValue(number.getValue().intValue()+1); // Sada: int + int
// Ali ovo još nije dobro: sada setValue dobiva int kao argument:
number.setValue(
    Integer.valueOf(number.getValue().intValue()+1)
); // OK.
```

- Ako trebamo dopustiti izmjenu objekata različitih tipova, tada bismo za svaki morali pisati zaseban wrapper
 - Vidi primjer i razrede `IntWrapper`, `DoubleWrapper`
- Posljedica:
 - Unosimo masovnu redundanciju kôda
 - Kršimo načela oblikovanja kvalitetnog kôda

■ Usporedimo razrede:

```
public class DoubleWrapper {  
  
    private Double value;  
  
    public DoubleWrapper() {  
        this.value = Double.valueOf(0.0);  
    }  
  
    public DoubleWrapper(Double value) {  
        this.value = value;  
    }  
  
    public Double getValue() {  
        return value;  
    }  
  
    public void setValue(Double value) {  
        this.value = value;  
    }  
  
    @Override  
    public String toString() {  
        if(value==null) return "";  
        return value.toString();  
    }  
}
```

```
public class IntWrapper {  
  
    private Integer value;  
  
    public IntWrapper() {  
        this.value = Integer.valueOf(0);  
    }  
  
    public IntWrapper(Integer value) {  
        this.value = value;  
    }  
  
    public Integer getValue() {  
        return value;  
    }  
  
    public void setValue(Integer value) {  
        this.value = value;  
    }  
  
    @Override  
    public String toString() {  
        if(value==null) return "";  
        return value.toString();  
    }  
}
```

- Moguće rješenje je izrada općenitog omotača koji se može primijeniti na bilo koji objekt
 - Definiramo razred `wrapper` koji omata primjerke razreda `Object`

■ Evo koda:

```
public class ObjectWrapper {  
    private Object value;  
  
    public ObjectWrapper(Object value) {  
        this.value = value;  
    }  
  
    public Object getValue() {  
        return value;  
    }  
  
    public void setValue(Object value) {  
        this.value = value;  
    }  
  
    @Override  
    public String toString() {  
        if(value==null) return "";  
        return value.toString();  
    }  
}
```

Primjer uporabe:

```
ObjectWrapper w =  
    new ObjectWrapper(new String("Ivana"));  
  
int len = ((String)w.getValue()).length();  
  
int vrijednost =  
    ((Integer)w.getValue()).intValue(); //???
```

- Moguće rješenje je izrada općenitog omotača koji se može primijeniti na bilo koji objekt
 - Definiramo razred `wrapper` koji omata primjerke razreda `Object`
 - Problem: prevoditelj na mjestu uporabe više ne zna kojeg je tipa doista objekt, pa moramo eksplicitno ukalupljivati
 - Naporno
 - Onemogućeno rano otkrivanje pogrešaka prilikom prevođenja

- Tehnologija Java Generics omogućava pisanje parametriziranih tipova (razreda, sučelja)
 - Pri definiciji razreda/sučelja u zagradama < i > definiraju se lokalni nazivi **koji u toj definiciji imaju ulogu tipa podatka**
 - Gdje god je potrebno napisati tip, navodi se to slovo
 - Primjer je prikazan na sljedećem slideu

Tehnologija Java Generics

```
public class Wrapper<T> {  
    private T value;  
    public Wrapper() {  
        this.value = null;  
    }  
    public Wrapper(T value) {  
        this.value = value;  
    }  
    public T getValue() {  
        return value;  
    }  
    public void setValue(T value) {  
        this.value = value;  
    }  
    @Override  
    public String toString() {  
        if(value==null) return "";  
        return value.toString();  
    }  
}
```

Definicija parametra

Uporaba definiranog parametra

- Pri stvaranju primjeraka parametriziranih razreda potrebno je za svaki parametar (zamjenu za tip) navesti konkretan tip koji u tom objektu predstavlja svaki od parametara
 - Opet se koriste zagrade
 - Pri deklaraciji tipa reference navodi se tip parametra
 - Pri pozivu konstruktora tip se može izostaviti (ali ne i zagrade)

Tehnologija Java Generics

```
Wrapper<Integer> iNumber = new Wrapper<Integer>(new Integer(10));
Wrapper<Double> dNumber = new Wrapper<Double>(new Double(15.0));
Wrapper<String> sWrapper = new Wrapper<String>("Super kul!!!");

String s1 = (String)sWrapper.getValue();    // Možemo kastati...
String s2 = sWrapper.getValue();            // Ne trebamo kastati!

int length = sWrapper.getValue().length(); // OK!

int broj = sWrapper.getValue().intValue(); // Compile error!
Integer i1 = (Integer)sWrapper.getValue(); // Compile error!

int res = iNumber.getValue()+1; // => int
String str = sWrapper.getValue()+1; // => String
```

Tehnologija Java Generics omogućava praćenje tipova tijekom prevođenja, ta se informacija ne prenosi u runtime.

Zahvaljujući tome prevoditelj dozvoljava da se eksplicitna ukalupljivanja ne pišu već sam prati kojeg je tipa koja vrijednost.

Tehnologija Java Generics

```
Wrapper<Integer> iNumber = new Wrapper<>(5);
Wrapper<Double> dNumber = new Wrapper<>(15.0);

System.out.println("Before: " + iNumber + ", " + dNumber);

updateRandomlyInt(iNumber);
updateRandomlyDouble(dNumber);

System.out.println("After: " + iNumber + ", " + dNumber);
```

Diamond-operator:
ako je jasan kontekst (primjerice, tip je naveden s lijeve strane), pri stvaranju objekta ne mora se ponavljati definicija tipa: ostaju samo zagrade

```
// Will compiler catch the error if we uncomment
// next line?

//updateRandomlyDouble(iNumber);
```

■ Implementacijski detalj:

- Parametriziranje je tehnologija koju koristi isključivo prevoditelj
- Parametri se pri generiranju byte-koda brišu i zamjenjuju razredom `Object` (ili granicama: više u nastavku)
 - Pazi: polimorfizam metoda (više metoda istog imena, različitih tipova argumenata) ne radi nad parametriziranim argumentima – formalno, to su sve `Object`:

```
void m(Wrapper<Integer> value) {...}  
void m(Wrapper<Double> value) {...}
```

- Uporabom parametriziranih tipova izbjegava se potreba za ukalupljivanjem
 - Prevoditelj prati što je kojeg tipa (do mjere u kojoj je to moguće tijekom prevođenja)
 - Stoga je sasvim legalno:

```
Wrapper<Integer> iNumber = new Wrapper<>(5);  
Wrapper<String> someText = new Wrapper("Hi there!");
```

```
System.out.println(iNumber.getValue().intValue());
```

```
System.out.println(someText.getValue().length());
```

Integer ima metodu
intValue()

String ima metodu
length()

■ Važno:

- Java ne podržava stvaranje polja parametriziranih tipova
 - `new Wrapper<String>[20]`
- Ako Vam to baš treba i ako sami garantirate da je sve OK, možete se poslužiti ukalupljivanjem između neparametriziranih i parametriziranih tipova (i potisnuti upozorenje prevodioca):
`Wrapper<String> array =
 (Wrapper<String>)new Wrapper[20];`
- Nije moguće stvoriti polje “čistog” parametra:
`T[] array = new T[30];`

- Na parametre se mogu postavljati ograničenja:
 - Ako parametar treba predstavljati bilo koji tip koji je razreda/sučelja R ili izveden iz njega, koristi se sintaksa T extends S :
`public class Wrapper<T extends Number> { ... }`
bi ograničio mogućnost primjene razreda `Wrapper` samo na brojeve
 - Prednost: tom je ograndom definiran i skup metoda koje objekt sigurno podržava pa ih se može pozivati

- Na parametre se mogu postavljati ograničenja:
 - Ako parametar treba predstavljati bilo koji tip koji je naviše razreda/sučelja `R` (drugim riječima, koji je u stablu nasljeđivanja od razreda `Object` pa do `R` ali ne ispod), koristi se sintaksa `T super S`:

```
public class Wrapper<T super Number> { ... }
```

bi ograničio mogućnost primjene razreda `Wrapper` do na primjerke razreda `Number` (ali ne `Integer`, `Double` i slično koji su iz njega izvedeni)
 - Vidjet ćemo na primjeru kolekcija kada ovo ima smisla: kolekciju tipa `T` može sortirati komparator bilo kojeg tipa koji je `X super T`.

- Na parametre se mogu postavljati ograničenja:
 - moguće je definirati i višestruka ograničenja koja se tada spajaju znakom &; primjerice neka je S razred a R i Q sučelja, možemo pisati:
`public class Wrapper<T extends S & R & Q> { ... }`
- Osim razreda, i metode mogu biti lokalno parametrizirane (neovisno o tome jesu li u parametriziranom razredu ili ne)
 - Parametar se definira prije povratne vrijednosti, npr.
`public <T> boolean m(T[] data, T element) {...}`
- Nestatičke metode vide parametar razreda i mogu ga koristiti u ogradi; statičke metode ne vide parametre razreda!

■ Primjer parametrizirane metode:

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

```
public static <T extends Comparable<T>> int  
countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
    for (T e : anArray)  
        if (e.compareTo(elem) > 0)  
            ++count;  
    return count;  
}
```

```
int b1 = countGreaterThan(new Integer[] {...}, Integer.valueOf(4));  
int b2 = countGreaterThan(new String[] {...}, "Jadranko");
```

Ideja: ako imam polje objekata tipa koji implementira sučelje `Comparable` parametrizirano tim istim tipom, znam da nad svakim elementom mogu pozvati metodu `compareTo`!

Primjeri:

```
class String implements  
Comparable<String> {...}
```

```
class Integer implements  
Comparable<Integer> {...}
```


- Parametriziranjem razreda nastaju novi tipovi podataka koji ne preuzimaju odnose parametara
 - Npr. Neka razred B nasljeđuje razred A
 - Za tip `wrapper` ne kažemo da nasljeđuje `wrapper<A>`; ta dva tipa su nepovezana i oba nasljeđuju samo `object`
- Uz fiksiran parametar, relacije između razreda i sučelja su u skladu s očekivanjem
 - Npr. Neka imamo parametrizirani razred B koji nasljeđuje parametrizirani A
 - `B<Integer>` je podtip od `A<Integer>` ali nije od `A<Number>`

Tehnologija Java Generics

```
public static void main(String[] args) {  
    Wrapper<Number> num1 = new Wrapper<>(new Integer(10));  
    Wrapper<Integer> num2 = new Wrapper<>(new Integer(10));  
    m1(num1);  
    m1(num2); ← Greška pri prevođenju;  
               m1 nije primjenjiva na Wrapper<Integer>  
    m2(num1);  
    m2(num2); ← OK  
               m2 je primjenjiva na Wrapper od bilo čega što je barem Number  
}  
  
static void m1(Wrapper<Number> num) {  
    System.out.println(num);  
}  
  
static <T extends Number> void m2(Wrapper<T> num) {  
    System.out.println(num);  
}
```

- Tehnologija Java Generics podržava još i zamjenske tipove
 - `<?>`,
 - `<? extends T>`,
 - `<? super T>`
- Nećemo ih dalje obrađivati