Objektno orijentirano programiranje

8: Generics. *Boxing* i *unboxing*

Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- https://creativecommons.org/licenses/by-nc-sa/4.0/

Motivacija za parametrizirane tipove (1/3)

- Napisati klasu za uređeni (nepromjenjivi) par cijelih brojeva.

08_Generics/…/example1/IntPair.java

```
package hr.fer.oop.generics.example1;
public class IntPair {
	private int first;
	private int second;
	public IntPair(int x, int y) {
		this.first = x;
		this.second = y;
	}
	public int getFirst() { return first; }
	public int getSecond() { return second; }
	@Override
	public String toString() {
		return "(" + first + ", " + second + ")";
	}
}
```

Motivacija za parametrizirane tipove (2/3)

- Napisati klasu za uređeni (nepromjenjivi) par stringova.
  - Crveno su označene promjene u odnosu na *IntPair*

08_Generics/…/example1/StringPair.java

```
package hr.fer.oop.generics.example1;
public class StringPair {
	private String first;
	private String second;
	public StringPair(String x, String y) {
		this.first = x;
		this.second = y;
	}
	public String getFirst() { return first; }
	public String getSecond() { return second; }
	@Override
	public String toString() {
		return "(" + first + ", " + second + ")";
	}
}
```

Motivacija za parametrizirane tipove (3/3)

- Jedine razlike u kodu su vezane za promjenu tipa
  - ostalo je ponavljanje istog koda
  - upotreba je također gotovo identična

08_Generics/…/example1/Main.java

```
package hr.fer.oop.generics.example1;
public static void main(String[] args) {
        IntPair ip = new IntPair(3, 5);
        System.out.println("First = " + ip.getFirst());
        System.out.println("Second = " + ip.getSecond());
        System.out.println("Pair = " + ip.toString());

        StringPair sp = new StringPair("A", "B");
        System.out.println("First = " + sp.getFirst());
        System.out.println("Second = " + sp.getSecond());
        System.out.println("Pair = " + sp.toString());
}
```

Korištenje reference tipa *Object* kao općenite reference (1/2)

- Možemo li konkretni tip zamijeniti klasom *Object*?
  - djelomično – *int* je primitivni tip, pa rješenje nije u potpunosti moguće, ali ignorirajmo tu činjenicu do daljnjeg

```
public class Pair {
	private Object first;
	private Object second;
	public Pair(Object x, Object y) {
		this.first = x;
		this.second = y;
	}
	public Object getFirst() { return first; }
	public Object getSecond() { return second; }
	@Override
	public String toString() {
		return "(" + first + ", " + second + ")";
	}
}
```

Korištenje reference tipa *Object* kao općenite reference (2/2)

- Spremanje u referencu tipa *Object* (upcast) uzrokuje naknadno eksplicitno ukalupljivanje (downcast)

```
Pair p = new Pair("A", "B");
String first = (String) p.getFirst();
String second = (String) p.getSecond();
```

- Veći problem je što prevodilac ne može provjeriti je li *downcast* ispravan
  - npr. sljedeći kod uzrokuje *ClassCastException*

```
Pair p = new Pair("A", new Food(…));
String second = (String) p.getSecond(); //ClassCastException
```

Parametrizirane klase i sučelja

- Pri definiciji klase/sučelja između znakova < i > definira se lokalni naziv za neki (u tom trenutku nepoznati, općeniti) tip koji želimo koristiti u klasi

08_Generics/…/example2/Pair.java

```
public class Pair<T> {
	private T first;
	private T second;
	public Pair(T x, T y) {
		this.first = x;
		this.second = y;
	}
	public T getFirst() { return first; }
	public T getSecond() { return second; }
	@Override
	public String toString() {
		return "(" + first + ", " + second + ")";
	}
}
```

Korištenje parametriziranih klasa (sučelja)

- Klasa *Pair* je parametrizirana, a *T* u *Pair<T>* predstavlja parametar parametriziranog tipa (engl. ***type parameter***)
- Možemo definirati par stringova, par artikala, par bilo čega
  - Konkretni tip se navodi prilikom korištenja i predstavlja argument parametriziranog tipa (engl. ***type argument***).
  - S desne strane izraza može se izostaviti argument i napisati samo <>, jer prevodilac iz deklaracije zaključuje koji se konkretni tip koristi

```
Pair<String> ps = new Pair<>("ABC", "DEF");
```

```
Pair<Food> pf = new Pair<>(new Food(...), new Food(...));
```

- Klase/sučelja se mogu parametrizirati po više tipova, npr. NekaKlasa<T, U, V>
  - način korištenja je isti

    NekaKlasa <String, Food, String> c = new NekaKlasa<>(…)

Primitivni tipovi i njihovi omotači

- Primitivni tipovi (*byte, short, int, long, char, boolean, float* i *double*) se ne mogu koristiti kao argumenti parametriziranog tipa!
- Ipak, postoje njihovi objektni omotači - klase *Byte, Short, Integer, Long, Character, Boolean, Float* i *Double*
  - klase koje u sebi sadrže (omataju) nepromjenjivi primitivni tip
  - dodatno nude i neke korisne statičke metode npr. *Double.toHexString(2.15), Integer.parseInt("343")* …
  - … i metode objekta koje služe za dohvat omotanog primitivnog tipa
  - moguće definirati npr. *Pair<Integer>*
- Napomena:
  - Klase *Byte, Short, Integer, Long, Float i Double* su izvedene iz klase *Number*
    - ovaj podatak će nam biti bitan u zadacima koji uskoro slijede

Boxing

- Postupak kojim se tip sa sistemskog stoga pretvara u objekt i stavlja na hrpu (engl. heap) naziva se ***boxing***.
  - Na hrpi se stvara novi objekt koji sadrži omotanu vrijednost (i informaciju o tipu objekta)
    - u Javi se primitivni tip pretvara u objekt omotač
    - u C#-u se to radi za vrijednosne tipove (engl. value type): int, struct…,
  - Kao rezultat nastaje referenca na nepromjenjivi omotani objekt

08_Generics/…/boxing_unboxing/BoxingUnboxing.java

```
int a = 10;
Integer x = Integer.valueOf(a); //boxing
```

Unboxing

- Dohvat omotane vrijednosti iz omotača naziva se ***unboxing***

08_Generics/…/boxing_unboxing/BoxingUnboxing.java

```
int a = 10;
Integer x = Integer.valueOf(a); //boxing
int c = x.intValue(); //unboxing
```

- Vrijednost omotana u objektu omotaču koji se nalazi na hrpi, kopira se na stog.

- Podsjetnik: Možemo napisati x = neštodrugo, ali to ne mijenja objekt unutar omotača, nego mijenja na što se referenca *x* odnosi
  - Također, možemo promijeniti a i c naknadno, jer su to memorijske lokacije na stogu i neće utjecati na omotanu vrijednost.

Automatski boxing i unboxing

- Javin prevodilac automatski generira kod za boxing i unboxing, pa možemo napisati

08_Generics/…/boxing_unboxing/BoxingUnboxing.java

```
int a = 10;
Integer x = a; //autoboxing: Integer.valueOf(a);
int c = x; //autounboxing: x.intValue();
```

- Podržane su i neke složenije situacije, npr.

```
int a = 10;
Integer x = Integer.valueOf(a); //boxing
int c = x.intValue(); //unboxing
int y = 20;
Integer z = x  + y;
// => (auto)unboxing x: x.intValue() + y  => produces int
// => (auto)boxing to z Integer.valueof(x.intValue() + y);
System.out.println("" + x + " + " + y + " = " + z);
// => uses StringBuilder to
// append strings and primitive values
```

Prednosti parametriziranih tipova

- Uporabom parametriziranih tipova izbjegava se potreba za ukalupljivanjem
- Prevodilac prati što je kojeg tipa (do mjere u kojoj je to moguće tijekom prevođenja), pa tako npr.

  Pair<String> p = new Pair<>("ABC", 123);

  uzrokuje pogrešku prilikom prevođenja zbog nepodudaranja drugog argumenta u konstruktoru (očekuje se String)
- Tip povratne vrijednosti poznat je unaprijed, npr.
  - *Pair<String> p* podrazumijeva da je *p.getFirst()* tipa *String*
    - može se napisati *p.getFirst().length()*
  - *Pair<Food> p* uzrokuje da je *p.getFirst()* tipa *Food*
    - može se napisati *p.getFirst().getPrice()* ali ne i *p.getFirst().length()*
  - Onemogućava slanje neispravnog argumenta nekoj metodi
    - Primjer: **08_Generics/…/example2/Main.java**

Kako Java interno podržava parametrizacije?

- U Javi je parametriziranje tehnologija koju koristi isključivo prevodilac (što npr. nije slučaj u C#-u)
- Parametri se pri generiranju *byte-koda* brišu i zamjenjuju klasom *Object* (ili granicama: više na kasnijim slajdovima)
  - dobiveni *byte-code* sadrži „normalne” klase, sučelja i metode
  - kako bi se osigurala tipska sigurnost prevodilac dodaje potrebna ukalupljivanja i eventualne potrebne dodatne metode u slučaju nasljeđivanja parametriziranih klasa
- Nije nešto o čemu se treba (previše) brinuti, ali za posljedicu ima činjenicu da preopterećivanje metoda ne radi nad parametriziranim tipovima
  - npr. nije moguće imati ove dvije metode u istoj klasi:

```
void m(Pair<Integer> value) {…}
void m(Pair<Double> value) {…}
```

Zloupotreba brisanja tipova

- Budući da Java briše tipove kod parametrizacije (engl. *type erasure*), navedeno se može zlouporabiti korištenjem tzv. *raw tipova*

```
Pair iPair = new Pair(5,5);
Pair sPair = new Pair("X", "Y");
```

  - prevodilac javlja upozorenje, ali prevodi kod
  - ovime se gubi tipska sigurnost i ovakav kod se ne preporuča
    - detaljniji primjer nalazi se u: **08_Generics/…/example2/RawPairMain.java**

Parametrizirane klase i nasljeđivanje

- Parametriziranjem klase nastaju novi tipovi podataka koji ne preuzimaju odnose parametara
  - npr. Ako je klasa *Food* izvedena iz *Item*, tada *Pair<Food>* nije izveden iz *Pair<Item>* i ne može se izvesti takvo ukalupljivanje
- Parametrizirane klase mogu naslijediti neku drugu klasu pri čemu se relacija nasljeđivanja čuva ako se koriste isti argumenti za parametrizaciju, npr. ako *Pair<T>* extends *Ntuple<T>* (tj. ako je par specijalni slučaj neke n-torke) tada je
  - Pair<String> je potklasa od Ntuple<String>
  - Pair<Food> je potklasa od Ntuple<Food>
  - Pair<Item> je potklasa od Ntuple<Item>
  - *Food extends Item,* ali *Pair<Food>* nije potklasa od *Ntuple<Item>*

Parametriziranje metoda (1)

- Ne-statičke metode mogu koristiti parametrizaciju klase, ali mogu biti i parametrizirane neovisno o parametrizaciji klase
  - Generički parametar se navodi prije povratnog tipa

08_Generics/…/example3/Pair.java

```
public class Pair<T> {
    ...
    public <V> void printWith(V another) {
        System.out.format("first: %s second %s %n",
                          this.toString(), another.toString());
    }
```

  - Kako i kod konstruktora, argument parametrizacije se ne mora navesti, ako je očit iz cijelog izraza, odnosno argumenata metode

08_Generics/…/example3/Main.java

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWith(sPair);
iPair.printWith("OOP");
```

Parametriziranje metoda (2)

- Prethodna parametrizacija metode nije pretjerano praktična, jer V može biti bilo što (*String*, *Food*, *Pair<String>*, *Number* …)
  - Jedino što prevodilac zna da je V izveden iz *Object* i jedine dostupne metode su one definirane klasom *Object*

08_Generics/…/example3/Pair.java

```
public class Pair<T> {
    ...
    public <V> void printWith(V another) {
        System.out.format("first: %s second %s %n",
                          this.toString(), another.toString());
    }
```

08_Generics/…/example3/Main.java

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWith(sPair);
iPair.printWith("OOP");
```

Parametriziranje metoda (3)

- Parametrizacija se može upotrijebiti i kao argument parametrizirane klase
  - U donjoj metodi navodimo da je *another* neki par (ne nužno istog tipa kao par nad kojim je metoda pozvana), pa možemo koristiti metode klase *Pair*

08_Generics/…/example3/Pair.java

```
public class Pair<T> {
    ...
    public <V> void printWithPair(Pair<V> another) {
        System.out.format("first: %s second %s,%s %n",
                this.toString(), another.getFirst().toString(),
                                 another.getSecond().toString());
    }
```

08_Generics/…/example3/Main.java

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWithPair(sPair);
iPair.printWithPair(iPair);
//iPair.printWithPair("OOP"); //compile error
```

Zamjenski tip prilikom parametriziranja

- U prethodnom primjeru metoda *printWithPoint* je parametrizirana tipom V, ali osim za propisivanje da je argument metode bio Point<V> nije bilo drugih varijabli ili argumenata koji bi koristili tip V niti se V koristio za tip povratne vrijednosti.
  - U takvim slučajevima moguće je koristiti zamjenski tip `?`

```
public class Pair<T> {
    ...
    public void printWithPair(Pair<?> another) {
        ...
    }
```

08_Generics/…/example3/Pair.java

  - primijetiti da ispred povratne vrijednost nema <?>
  - također, nije moguće `void <?> printWith(? another)`

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWithPair(sPair);
//iPair.printWithPair("OOP"); //compile error
```

08_Generics/…/example3/Main.java

Statičke metode i parametrizacija

- Statički članovi i statičke metode ne mogu kao argument ili kao lokalnu varijablu koristiti parametrizaciju klase, ali mogu imati vlastitu
- Klase ne mogu imati statičke varijable parametriziranog tipa

```
public class Pair<T> {
    static T t1; //compile error
    ...
    public static <V> void someMethod(Pair<V> arg) {
       V v; //OK
       T t; //compile error
        ...
    }
```

Compile error: *Cannot make a static reference to the non-static type T*

Ograničenja na tip parametra prilikom parametrizacije

- Parametar parametriziranog tipa može biti ograničen (donja i gornja ograda)
- Gornja ograda (engl. *upper bound*): T extends R
  - T se može ukalupiti u R
    - T je R, T potklasa od R (direktno ili indirektno), ili T ili T-ove bazne klase implementiraju R
  - npr. *Pair<T extends Number>* znači da možemo definirati par tipa Integer, par tipa Float, … pa za posljedicu ima da je *getFirst()* sigurno izveden iz *Number* (t.j. može se spremiti u referencu tipa *Number*)
- višestruka ograničenja se spajaju znakom &
  - class SomeClass<T extends S & R & Q> { … }
  - T se može ukalupiti u S, R i Q. Jedan ili nijedan od tipova S, R i Q može biti klasa, a ostali su sučelja

Korištenje gornje ograde – primjer 1 (1/2)

- Podsjetnik: *Integer extends Number*
  - Klasa *Number* je apstraktna, ali izraz

    Pair<Number> num1 = new Pair<>(10, 5);

    je valjan zbog *autoboxinga* iz *int* u *Integer*, a *Integer* „je” *Number*
  - Pair<Integer> nije potklasa od Pair<Number> pa komentirani redak nije sintaksno ispravan

```
public static void main(String[] args) {
        Pair<Number> num1 = new Pair<>(10, 5);
        Pair<Integer> num2 = new Pair<>(10, 3);
        m1(num1); //OK – num1 je Pair<Number>
        //m1(num2); //compile error
        ...
}
static void m1(Pair<Number> num) {
        System.out.println(num);
}
```

08_Generics/…/example4/Main.java

Korištenje gornje ograde – primjer 1 (2/2)

- m2 prima par bilo kojeg tipa koji se može ukalupiti u *Number*
  - Posljedično možemo poslati *Pair<Integer>, Pair<Number>*, ali ne i *Pair<String>*

```
public static void main(String[] args) {
        Pair<Number> num1 = new Pair<>(10, 5);
        Pair<Integer> num2 = new Pair<>(10, 3);
        ...
        m2(num1);
        m2(num2);

        Pair<String> ps = new Pair<>("A", "B");
        //m2(ps); //compile error
}
static <T extends Number> void m2(Pair<T> num) {
        System.out.println(num);
}
```

08_Generics/…/example4/Main.java

Korištenje gornje ograde – primjer 2 (1/2)

- Napisati metodu koja će prebrojati koliko je elemenata polja veće od nekog zadanog elementa
- Isti algoritam bez obzira na tip polja. Jedini uvjet je da se elementi mogu uspoređivati.
- Prva ideja za deklaraciju metode

```
public static <T> int cnt(T[] data, T x) {…}
```

  - Što prevodilac zna o T-u?
    - Ništa osim da je izveden iz *Object* pa se mogu koristiti samo metode iz klase *Object*
- Potrebno postaviti ograničenje da su elementi iz T međusobno usporedivi, za što je predviđeno sučelje *Comparable<T> s* metodom *compareTo*

Korištenje gornje ograde – primjer 2 (2/2)

- Metodu možemo koristiti s poljem stringova, poljem cijelih brojeva, ali ne i poljem artikala

```
public static void main(String[] args) {
    Integer[] a = {1, 2, 3, 4, 5, 6};
    System.out.println(countGreaterThan(a, 2));
    String[] b = {"MAT3R", "OOP", "OS", "TZK", "ELE1", "FIZ2"};
    System.out.println(countGreaterThan(b, "OOP"));
}
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;
       }
```

08_Generics/…/example5/Main.java

Zamjenski tipovi i donja ograda ograničenja

- Donja ograda se navodi korištenjem riječi *super*
  - npr. *void <T super Beverage> method(T arg)* znači da argument arg mora biti *Beverage* ili neka nadređena klasa (*Item*, *Object*)
- Zamjenski tip se označava s upitnikom i može biti korišten i sa *super* i s *extends*
  - <? extends E> (skoro) isto kao <T extends E>
  - <? super E>

- Zamjenski tipovi i donje ograde će se intenzivnije koristiti u nekim od sljedećih predavanja
  - Za znatiželjne i naprednije studente detaljnije o zamjenskim tipovima prilikom parametrizacije metoda može se naći na :

https://docs.oracle.com/javase/tutorial/extra/generics/methods.html
https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

Ograničenja tehnologije Java Generics (1)

- Nije moguće direktno stvoriti objekt tipa parametra parametrizirane klase

```
class Gen<T> {
  T obj;
  void test(){
    obj = new T(); //nije moguće
  }
}
```

- Podsjetnik: Parametrizirati se može samo po referencijskom tipu, ali ne po primitivnom tipu, npr. `Gen<int>` nije moguće, ali `Gen<Integer>` jest

Ograničenja tehnologije Java Generics (2)

- Moguće je deklarirati polje parametriziranog tipa ili polje parametriziranih klasa, ali se ne može direktno stvoriti

```
class Gen<T> {
  T[] arr; //O.K.
  Gen(int size){
    arr = new T[size]; //nije moguće
    Gen<Integer>[] gens = new Gen<Integer>[20]; //nije moguće
```

- Postoje neka zaobilazna rješenja za koja prevodilac javlja upozorenja, što možemo isključiti označavajući metodu ili varijablu oznakom `@SuppressWarnings("unchecked")`

```
Gen<?>[] gens = new Gen<?>[20];
arr = (T[]) new Object[size];
Gen<Integer>[] gens = (Gen<Integer>[]) new Gen<?>[20];
```

  - Napomena: Korištenje ograničenja komplicira prethodni kod, pa tako umjesto new Object[size] pišemo new Bound[size] itd…

Ograničenja tehnologije Java Generics (3)

- Statički članovi i statičke metode ne mogu koristiti parametrizaciju klase, ali mogu imati vlastitu
  - Moguće koristiti isti naziv tipa, ali se ne preporuča

```
class Gen<T> {
  static T obj; //compile error
  static T test(){  //compile error
    ...
  }
  static <V> V test(V arg, T arg) { //compile error
    T t; //compile error
  }
  static <V extends Number> V test(V arg) { V v; } //OK

  static <T> void test3(T x) { //neki drugi T
    T t; //OK – ovo nije T s kojim je klasa parametrizirana
  }
```

Zadaci za vježbu (1/2)

- Preraditi zadatak 2 iz *6-Rekapitulacija* tako da klasa MyList predstavlja jednostruko povezanu listu parametriziranu po nekom tipu.

08_Generics/…/task1.*

Zadaci za vježbu (2/2)

- Izmijeniti klasu *Pair* tako da prvi i drugi element para mogu biti različitog tipa. Nakon toga napisati parametriziranu klasu *Point* koja predstavlja točku u 2D prostoru, i ograničiti parametrizaciju samo na tipove izvedene iz klase *Number*.

08_Generics/…/task*

- Napisati metodu koja će naći težište svih točaka spremljenih u listi tipa *MyList*.
  - Metoda mora raditi s bilo kojom listom točaka numeričkog tipa, dakle i s *MyList<Point<Integer>>* i s *MyList<Point<Double>>*
- Izmijenite prethodni zadatak tako da sadržaj liste mogu činiti točke različitog numeričkog tipa
  - *Npr. u listi se nalaze istovremeno i Point<Integer> i Point<Double>, ali u listi ne može biti npr. samo Integer.*