



CS202 - OBJEKTNO-ORIJENTISANO PROGRAMIRANJE 2

Generičke klase i metodi

Lekcija 06

PRIRUČNIK ZA STUDENTE

CS202 - OBJEKTNO-ORIJENTISANO PROGRAMIRANJE 2

Lekcija 06

GENERIČKE KLASSE I METODI

- ✓ Generičke klase i metodi
- ✓ Poglavlje 1: Šta su generičke klase?
- ✓ Poglavlje 2: Korist od primene generičkih klasa i metoda
- ✓ Poglavlje 3: Definisanje generičkih klasa i interfejsa
- ✓ Poglavlje 4: Generički metodi
- ✓ Poglavlje 5: Studija slučaja: Sortiranje niza elemenata
- ✓ Poglavlje 6: Sirovi tipovi i kompatibilnost unazad
- ✓ Poglavlje 7: Džoker generički tipovi
- ✓ Poglavlje 8: Brisanje tipova
- ✓ Poglavlje 9: Ograničenja primene generičkih tipova
- ✓ Poglavlje 10: Vežba – Pokazni primeri
- ✓ Poglavlje 11: Vežba – Zadaci za individualni rad
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Generičke (uopštene) klase i metodi omogućuju vam da otkrijete greške u vreme kompilacije programa, umesto u vreme njegovog izvršenja.

Ova lekcija treba da ostvari sledeće ciljeve:

- Da opiše prednosti primene generičkih klasa
- Da ukaže na upotrebu generičkih klasa i interfejsa
- Da definiše generičke klase i interfejse
- Da objasni zašto generički tipovi mogu da poboljšaju pouzdanost i čitljivost
- Da definišu i ukažu na upotrebu generičkih metoda i graničnih generičkih tipova
- Da razvije generički metod sortiranja radi sortiranja reda **Comarable** objekata
- Da ukaže na upotrebu tipova radova za povratnu kompatibilnost
- Da objasni zašto su potrebni džoker generički tipovi
- Da opiše generički tip za brisanje i da izlista određena ograničenja generičkih tipova prouzrokovanih tipom za brisanje.
- Ukaže na projektovanje i implementaciju generičkih klasa za matrice

Referenca: Y. Daniel Liang, INTRODUCTION TO JAVA PROGRAMMING (COMPREHENSIVE VERSION), Tenth Edition, Pearson, ISBN 10: 0-13-376131-2, ISBN 13: 978-0-13-376131-3

Ovo je osnovni udžbenik za ovaj predmet i preporučuje se studentima da ga koriste,

▼ Poglavlje 1

Šta su generičke klase?

POJAM GENERIČKIH KLASA I METODA

Generičke (opšte) klase i metode vam omogućuju parametrizovanje tipova.

Generičke (opšte) klase i metode vam omogućuju parametrizovanje tipova. To znači da možete da definišete neku klasu ili metod sa generičkim (opštim) tipom koje kompajler može da zameni sa konkretnim tipovima. Na primer, Java definiše generičku klasu **ArrayList** za sortiranje elemenata generičkog (opšteg) tipa. Sa ovom generičkom klasom, možete da kreirate **ArrayList** objekat koji sadrži stringove (nizove sa znacima) i **ArraList** objekat koji sadrži brojeve. U ovom slučaju, stringovi i brojevi su konkretni tipovi koji zamenjuju generičke (opšte) tipove.

Glavna korist od generičkih klasa i metoda je u pronalaženju grešaka u vreme kompilacije umesto u fazi izvršenja programa. Generička klasa ili metod dozvoljava vam da specificirate dozvoljene tipove objekata sa kojima klasa ili metod može da radi. Ako pokušate da upotrebite neki nekompatibilan objekat (tipa koji nije specificiran), kompajler će otkriti grešku.

Na primer, neka konkretna klasa Box izgleda ovako:

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

Genreički (opšti) oblik klase Box bi izgledao ovako:

```
public class GenericBox<T> {  
    // T je skraćenica od "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

Koja je razlika između ove dve klase?

U klasi **Box** koristi se atribut object čiji je tip Object.

U klasi **GenericBox** koristi se atribut `t` tipa **T**, gde se oznaka za tip **T** može da odnosi i na tip `Object`, ali i za neke druge tipove. Zato je klasa **GeneriBog** opštija, tj. ona je generička.

ZAŠTO KORISTITI GENERIČKE TIPOVE?

Generici omogućavaju da tipovi (klase i interfejsi) postanu parametri pri definisanju klasa, interfejsa ili metoda. Omogućavaju vam da ponovo koristite isti kod sa drugim ulazima

Generici omogućavaju da tipovi (klase i interfejsi) postanu parametri pri definisanju klasa, interfejsa ili metoda. Tipovi kao parametri, ili parametri tipova omogućavaju vam da ponovo koristite isti kod sa drugim ulazima. Dok je kod korišćenja formalnih parametara, to su vrednosti, dok kod unosa parametara tipova, to su tipovi (klase ili interfejsi).

Kod koji upotrebljava generike donosi niz koristi u odnosu na kodove bez generika:

1. **Bolja provera tipa u vreme kompilacije:** Java kompajler primenjuje čvrstu proveru tipa generičkog koda i javlja greške ako kod ne poštuje sigurnost tipa. Otklanjanje grešaka u vreme kompilacije je jednostavnije nego u vreme izvršenja programa, kada je teško naći grešku.
2. **Eliminacija konverzije tipa:** Sledeći kod bez generika zahteva konverziju tipa (**casting**),

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

: Upotrebom generika, programeri mogu da primene generičke algoritme koji koriste kolekcije različitih tipova, koji mogu da se prilagode, i koje sigurni tipovi i lakše se čitaju.

Ubacivanjem u program generike, konverzija tipova nije više potrebna:

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);    // bez konverzije
```

▼ Poglavlje 2

Korist od primene generičkih klasa i metoda

ZAŠTO SE KORISTE GENERIČKE KLASSE I METODI?

Razlog upotrebe Java generičkih klasa i metoda je u otkrivanju grešaka u vreme kompilacije.

Od izdanja JDL 1.5 Java nam omogućava korišćenje generičkih klasa i metoda. Na slici 1 prikazana je promena interfejsa **java.lang.Comparable** koja je usledila pojavom JDK 1.5.

<pre>package java.lang; public interface Comparable { public int compareTo(Object o) }</pre>	<pre>package java.lang; public interface Comparable<T> { public int compareTo(T o) }</pre>
a) Pre JDK 1.5	a) Od JDK 1.5

Slika 2.1 Interfejs Comparable pre i posle pojave JDK 1.5 [1]

Na slici 1.a, oznaka **<T>** predstavlja formalni generički (opšti) tip, koji se kasnije može zameniti stvarnim, konkretnim tipom. Zamena generičkog tipa sa konkretnim tipom naziva se *generička konkretizacija* (engl., **generic instantiation**). Konvencijom, formalni generički tip se označava sa velikim slovom **E** ili **T**.

Radi ilustracije koristi od primene generike, pogledajmo kod na slici 2. Iskaz na slici 2a deklarise **c** kao referentnu promenljivu čiji je tip **Comparable** i pobuđuje metod **compareTo** da bi uporedio objekat **Date** sa jednim stringom. Kod se ovaj kod kompilira, ne javlja se grešae, ali pri izvršenju dolazi do greške jer se string ne može upoređivati sa datumom, tj. sa **Date**,.

<pre>Comparable c = new Date(); System.out.println(c.compareTo("red"));</pre>	<pre>Comparable<Date> c = new Date(); System.out.println(c.compareTo("red"));</pre>
a) Pre JDK 1.5	b) Posle JDK 1.5

Slika 2.2 Novi generički tip otkriva moguće greške u vreme kompilacije [1]

Iskaz na slici 2b objavljuje da je **c** referentna promenljiva čiji je tip **Comparable<Date>** i koji pobuđuje metod **compareTo** radi upoređenja objekta **Date** sa jednim stringom. Ovaj kod generiše grešku pri kompilaciji, jer je prenet argument metodom **compareTo** mora da bude tipa **Date**. Kako je greška otkrivena pri kompilaciji, a ne u fazi izvršenja, jasno je da je generički tip doprineo većoj pouzdanosti programa.

PRIMER KLASSE ARRAYLIST

Konverzija tipova podataka (engl. casting) nije potrebna da bi se dobila vrednost iz liste koja ima elemente određenog tipa, jer kompajler već zna tip elementa

UML dijagram klase **ArrayList** data je na slici 3., i to u verziji pre JDK 1.5 i od verzije JDK 1.5. Od verzije JDK 1.5 klasa **ArrayList** je generička klasa.

java.util.ArrayList	java.util.ArrayList<E>
<pre> +ArrayList() +add(o: Object): void +add(index: int, o: Object): void +clear(): void +contains(o: Object): boolean +get(index: int): Object +indexOf(o: Object): int +isEmpty(): boolean +lastIndexOf(o: Object): int +remove(o: Object): boolean +size(): int +remove(index: int): boolean +set(index: int, o: Object): Object </pre>	<pre> +ArrayList() +add(o: E): void +add(index: int, o: E): void +clear(): void +contains(o: Object): boolean +get(index: int): E +indexOf(o: Object): int +isEmpty(): boolean +lastIndexOf(o: Object): int +remove(o: Object): boolean +size(): int +remove(index: int): boolean +set(index: int, o: E): E </pre>
a) Pre JDK 1.5	b) Posle JDK 1.5

Slika 2.3 Klasa ArrayList koja je generička klasa od JDK 1.5 [1]

Na primer, sledeći iskaz kreira listu tekstova (strings):

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

U nju možete dodati samo tekstovne elemente (**strings**), kao na primer:

```
list.add("Red");
```

Ako bi dodali element nekog drugog tipa podataka (npr. ceo broj), kompajler bi javio grešku.

Generički tipovi moraju da budu tipovi sa referencama. Yato se ne mogu zameniti sa primitivnim tipovima (int, double ili char).

Sledeći iskaz je zbog toga pogrešan:

```
ArrayList<int> intList = new ArrayList<>();
```

Ispravno je napisati:

```
ArrayList<Integer> intList = new ArrayList<>();
```

Ako zatim napišete: `intList.add(5)` Java automatski konvertuje broj 5 u `new Integer(5)`. To se zove **autoboksing** (engl. **autoboxing**).

Konverzija tipova podataka (engl. **casting**) nije potrebna da bi se dobila vrednost iz liste koja ima elemente određenog tipa, jer kompajler već zna tip elementa. Na primer, sledeći iskaz kreira listu koja sadrži elemente tekstova (**strings**), dodaje nove u listu i čita tekstovne elemente iz listi:

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Red");
3 list.add("White");
4 String s = list.get(0); // Nema potrebe za konverzijom
```

Pre korišćenja JDK 1.5, bez upotrebe generika, morali bi da izvršite konverziju povratne vrednosti u String, jer: je:

```
String s = (String)(list.get(0)); // Neophodna konverzija pre JDK 1.5
```

AUTOANBOKSING - DODELJIVANJE TIPRA PRIMENOM LISTE SA UČAURENIM ELEMENTIMA

Autoanboksing je direktno dodeljivanje tipa nekoj promenljivoj primitivnog tipa ako su elementi liste ućaureni tipovi (wrapper types).

Ako su elementi liste tzv. ućaureni tipovi (eng. **wrapper types**), kao što su **Integer**, **Double**, **i Character**, onda možete da direktno dodelite tip nekoj promenljivoj primitivnog tipa. To se zove autoanboksing (engl. **autounboxing**). Na primer, pogledajte sledeći kod:

```
ArrayList<Double> list = new ArrayList<Double>();
list.add(5.5); // 5.5 je konvertovano u new Double(5.5)
list.add(3.0); // 3.0 je konvertovano u new Double(3.0)
Double doubleObject = list.get(0); // Nije potrebna konverzija
double d = list.get(1); // Automatska konverzija u double
```

U linijama 2 i 3, 5.5 i 3.0 su automatski konvertovane u **Double** objekte i dodati listi. U liniji 4, prvi element liste je promenljiva tipa **Double**. Nije potrebna konverzija (casting) jer je list lista deklarirana sa **Double** objektima. U liniji 5, drugi element u listi list je promenljiva tipa double. **Objekat** u **list.get(1)** je automatski konvertovan u jednu u vrednost primitivnog tipa.

ZADACI ZA SAMOSTALNI RAD

Proverite vaše razumevanje generičkih klasa i metoda

3. Šta je pogrešno u iskazima (a)? Da li je kod u (b) korektno napisan?

1. Da li ima grešaka pri kompajliranju iskaza (a) i (b)?


```
ArrayList dates = new ArrayList();  
dates.add(new Date());  
dates.add(new String());
```

a) Pre JDK 1.5

```
ArrayList<Date> dates = new  
    ArrayList<Date>();  
dates.add(new Date());  
dates.add(new String());
```

b) Posle JDK 1.5

Slika 2.4 Provera grešaka pri kompilaciji programa [1]

2. Šta je pogrešno u slučaju (a)? Da li je kod u slučaju (b) ispravan?

```
ArrayList dates = new ArrayList();  
dates.add(new Date());  
Date date = dates.get(0);
```

a) Pre JDK 1.5

```
ArrayList<Date> dates = new  
    ArrayList<Date>();  
dates.add(new Date());  
Date date = dates.get(0);
```

b) Posle JDK 1.5

Slika 2.5 Provera ispravnosti napisanog programskog koda [1]

2. Šta je pogrešno u iskazima (a)? Da li je kod u (b) korektno napisan?

▼ Poglavlje 3

Definisanje generičkih klasa i interfejsa

KLASA GENERICSTACK

Generički tip se definiše za klasu ili interfejs. Kada se klasa koristi pri kreiranju objekta, mora da se koristi i konkretan tip ili kada se koristi klasa ili interfejs za deklarisanje referentne prom

Na slici 1 prikazana je GenericStack klasa koja obezbeđuje stek memoriju i obezbeđuje operacije za manipuliranje stekom.

GenericStack <E>	
-list: java.util.ArrayList<E>	Lista u koju se smeštaju elementi
+GenericStack() +getSize(): int +peek(): E +pop(): E +push(o: E): void +isEmpty(): boolean	Kreira prazan stek Vraća broj elemenata u ovom steku Vraća element na vrhu u ovom steku Vraća i uklanja element na vrhu u ovom steku Dodaje novi element na vrh ovog steka Vraća istiniti ako je stek prazan

Slika 3.1 Klasa GenericStack [1]

Listing klase **GenericStack** izgleda ovako:

```
public class GenericStack<E> {  
    private java.util.ArrayList<E> list = new java.util.ArrayList<>();  
  
    public int getSize() {  
        return list.size();  
    }  
  
    public E peek() {  
        return list.get(getSize() - 1);  
    }  
  
    public void push(E o) {  
        list.add(o);  
    }  
  
    public E pop() {  
        E o = list.get(getSize() - 1);  
        list.remove(getSize() - 1);  
        return o;  
    }  
}
```

```
public boolean isEmpty() {  
    return list.isEmpty();  
}  
  
@Override  
public String toString() {  
    return "stack: " + list.toString();  
}  
}
```

Sledeći primer kreira stek za smeštaj stringova i dodaje tri stringa u stek:

```
GenericStack<String> stack1 = new GenericStack<>();  
stack1.push("London");  
stack1.push("Paris");  
stack1.push("Berlin");
```

Sledeći primer prikazuje kreiranje steka za smeštaj celih brojeva i način dodavanja tri cela broja u stek.

```
GenericStack<Integer> stack2 = new GenericStack<>();  
stack2.push(1); // autoboksing 1 u new Integer(1)  
stack2.push(2);  
stack2.push(3);
```

Umesto da koristite generički tip, mogli ste koristiti element tipa **Object**, koji može da predstavi bilo koji tip objekta. Međutim, upotrebom generičkog tipa popravljaju se pouzdanost i čitljivost softvera, jer se pojedine greške mogu da otkriju prilikom kompilacije, umesto prilikom izvršenja. Na primer, u slučaju objekta `stack1`, korišćena je deklaracija **`GenericStack<String>`**, samo se stringovi mogu smeštati u stek. Ako bi se smeštao ceo broj, na primer, prilikom kompilacije javila bi se greška.

▼ Poglavlje 4

Generički metodi

KLASA GENERICMETHODDEMO

Upotrebom generičkih tipova definišu se generički metodi

Upotrebom generičkih tipova definišu se generički metodi. Na slici 1 prikazan je listing generičkog metoda print (linije 10-14) radi štampanja reda objekata. Linija 6 prebacuje red objekata celih brojeva pri pobuđivanju generičkog metoda **print**. Linija 7 pobuđuje metod print sa redom stringova.

```
public class GenericMethodDemo {  
    public static void main(String[] args ) {  
        Integer[] integers = {1, 2, 3, 4, 5};  
        String[] strings = {"London", "Paris", "New York", "Austin"};  
  
        GenericMethodDemo.<Integer>print(integers);  
        GenericMethodDemo.<String>print(strings);  
    }  
  
    public static <E> void print(E[] list) {  
        for (int i = 0; i < list.length; i++)  
            System.out.print(list[i] + " ");  
        System.out.println();  
    }  
}
```

Da bi deklarirali generički metod, treba da postavite generički tip <E> odmah posle ključne reči static u zaglavlju metoda:

```
public static <E> void print(E[] list)
```

Da bi se prizvao generički metod, treba navesti ime metoda sa stvarnim tipom u uglastim zagradama:

```
GenericMethodDemo.<Integer>print(integers);  
GenericMethodDemo.<String>print(strings);
```

ili kraće:

```
print(integers);  
print(strings);
```

U ovom drugom slučaju, stvarni tip nije eksplicitno naveden. Kompajler automatski otkriva stvarni tip.

.

OGRANIČEN GENERIČKI TIP

Ograničen generički tip je generički tip koji se može specificirati i kao pod-tip nekog tipa.

Generički tip se može specificirati i kao pod-tip nekog tipa. Ovakav generički tip se naziva ograničen (engl. **bounded**). Listing prikazuje metod **equalArea()** a ograničen generički tip **<E extends GeometricObject>** (linija 7) specificira da je **E** generički pod-tip od **GeometricObject**. Da bi prizvali metod **equalArea()**, treba da prenesete dva objekta tipa **GeometricObject**.

```
public class BoundedTypeDemo {  
    public static void main(String[] args ) {  
        Rectangle rectangle = new Rectangle(2, 2);  
        Circle circle = new Circle(2);  
  
        System.out.println("Same area? " +  
            equalArea(rectangle, circle));  
    }  
  
    public static <E extends GeometricObject> boolean equalArea(  
        E object1, E object2) {  
        return object1.getArea() == object2.getArea();  
    }  
}
```

Neograničen generički tip **E** je isto što i **<E extends Object>**

Da bi se definisao generički tip neke klase, treba ga postaviti posle naziva klase, kao na primer: **GenericStack<E>**.

Da bi se definisao generički tip za metod, treba postaviti generički tip pre povratnog tipa metoda, kao na primer: **<E> void max(E o1, E o2)**.

VIDEO: GENERIČKI METODI

Intermediate Java Tutorial - 17 - Generic Methods (4,38 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: PRIMENA GENERIČKIH METODA

Intermediate Java Tutorial - 18 - Implementing a Generic Method (4,56 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: GENERIČKI POVRATNI TIPOVI

Intermediate Java Tutorial - 19 - Generic Return Types (6,04 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 5

Studija slučaja: Sortiranje niza elemenata

SORTIRANJE NIZA ELEMENATA

Možete da razvijete generički metod za sortiranje niza Comparable objekata.

Ovde ćemo navesti generički metod za sortiranje objekata reda **Comparable** objekata. Objekti su primeri interfejsa **Comparable**, a upoređuju se primenom metoda **compareTo()**. Da bi se metod testirao, program sortira niz celih brojeva, niz brojeva duple preciznosti, niz karaktera i niz stringova (tekstualnih elemenata). Program je prikazan na slici 1, dobijeni rezultati, tj. prikaz sortiranih nizova koje su kreirani u klasi GenericSort. Slika 1 prikazuje dobijene rezultate sortiranja.

```
Sorted Integer objects: 2 3 4  
Sorted Double objects: -22.1 1.3 3.4  
Sorted Character objects: J a r  
Sorted String objects: Kim Susan Tom
```

Slika 5.1 Prikaz rezultata sortiranja [1]

Metod **sort()** u ovom programu sortira niz sa bilo kojim tipom objekata, pod uslovim da taj je tip podržan sa interfejsom **Comparable**. Generički tip je definisan sa **<E extends Comparable<E>>** (linija 36). Ovo ima dva značenja. Prvo, specificira da je **E** pod-tip od **Comparable**. Drugo, specificira da elementi koji se upoređuju su takođe tipa **E**.

Metod **sort()** upotrebljava metod **compareTo** da bi odredio redosled objekata u nizu (linija 46). **Integer, Double, Character i String** primenjuju **Comparable**, te objekti ovih klasa se mogu upoređivati sa metodom **compareTo**.

Program kreira redove **Integer** objekata, **Double** objekata, **Character** objekata i **String** objekata (linije 4-16) i poziva **sort()** metod radi sortiranja ovih redova (linije 19-22).

```
public class GenericSort {  
    public static void main(String[] args) {  
        // Kreiranje niza sa Integer elementima  
        Integer[] intArray = {new Integer(2), new Integer(4),  
                               new Integer(3)};  
  
        // Kreiranje niza sa Double elementima  
        Double[] doubleArray = {new Double(3.4), new Double(1.3),  
                                 new Double(-22.1)};
```

```
// Kreiranje niza Character elementima
Character[] charArray = {new Character('a'),
    new Character('J'), new Character('r')};

// Kreiranje niza sa String elementima
String[] stringArray = {"Tom", "Susan", "Kim"};

// sortiranje nizova
sort(intArray);
sort(doubleArray);
sort(charArray);
sort(stringArray);

// Prikaz sortiranih nizova
System.out.print("Sorted Integer objects: ");
printList(intArray);
System.out.print("Sorted Double objects: ");
printList(doubleArray);
System.out.print("Sorted Character objects: ");
printList(charArray);
System.out.print("Sorted String objects: ");
printList(stringArray);
}

/** Sortiranje niza sa uoredljivim (Comparable) elementima */
public static <E extends Comparable<E>> void sort(E[] list) {
    E currentMin;
    int currentMinIndex;

    for (int i = 0; i < list.length - 1; i++) {
        // Nalaženje minimuma u listi list[i+1..list.length-2]
        currentMin = list[i];
        currentMinIndex = i;

        for (int j = i + 1; j < list.length; j++) {
            if (currentMin.compareTo(list[j]) > 0) {
                currentMin = list[j];
                currentMinIndex = j;
            }
        }

        // Zamena list[i] sa list[currentMinIndex] ako je potrebno;
        if (currentMinIndex != i) {
            list[currentMinIndex] = list[i];
            list[i] = currentMin;
        }
    }
}

/** Štampanje niza objekata */
public static void printList(Object[] list) {
    for (int i = 0; i < list.length; i++)
```



```
        System.out.print(list[i] + " ");  
        System.out.println();  
    }  
}
```

▼ Poglavlje 6

Sirovi tipovi i kompatibilnost unazad

PRIMER KORIŠĆENJE SIROVOG TIPRA

Generička klasa ili interfejs koji se koristi bez specifikacije konkretnog tipa, a koji se naziva sirovi tip, omogućava kompatibilnost unazad, tj. kompatibilnost sa ranijim verzijama Java.

Možete da upotrebite generičku klasu bez specifikacije konkretnog tipa, kao na ovom primeru:

```
GenericStack stack = new GenericStack(); // sirov tip
```

Ovo je ekvivalentno sa iskazom (po posledicama):

```
GenericStack<Object> stack = new GenericStack<Object>();
```

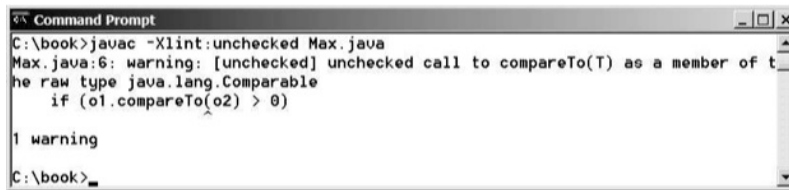
Sirovi tip: Generička klasa koje se koriste bez tipa parametra, nazivaju se sirovim tipom (engl., **raw type**). Upotrebom sirovih tipova omogućava se kompatibilnost unazad, tj. sa prethodnim verzijama Java. Na primer, generički tip se koristi u `java.lang.Comparable` od JDK 1.5 ali još puno programa koristi sirovi tip **Comparable**, kao sedeći primer.

```
1 public class Max {
2     /** Vraća maksimum od dva objekta */
3     public static Comparable max(Comparable o1, Comparable o2) {
4         if (o1.compareTo(o2) > 0)
5             return o1;
6         else
7             return o2;
8     }
9 }
```

Comparable o1 i **Comparable o2** su sirovi tipovi deklaracija. One nisu sigurne. Evo primera:

```
Max.max("Welcome", 23); // 23 ije autboksovan u s autoboxed into new Integer(23)
```

Ovo bi izazvalo grešku pri izvršenju programa, jer se ne može upoređivati **String** sa objektom celog broja (**Integer**). Java kompajler bi pokazao tu grešku u vidu poruke date na slici.



Slika 6.1 Poruka o grešci u izvršenju programa [1]

POBOLJŠANA KLASA MAX

Primenom generičkog tipa u metodu `max()` dobija se prikaz greške prilikom kompilacije, ako oba tipa dva argumenta nisu istog tipa.

Bolji način za pisanje metoda **max** je da se koristi generički tip, je sledeći:

```
public class Max {  
    /** Vraća maksimum od dva objekta */  
    public static Comparable max(Comparable o1, Comparable o2) {  
        if (o1.compareTo(o2) > 0)  
            return o1;  
        else  
            return o2;  
    }  
}
```

Ako pozovete metod **max** upotrebom instrukcije:

```
// 23 je outboksovan u new Integer(23)  
MaxUsingGenericType.max("Welcome", 23);
```

dobićete prikaz greške kompilacije, jer dva argumenta u metodu **max** moraju biti istog tipa (npr. dva stringa ili dva objekata celih brojeva (integer). Pored toga, tip **E** mora biti podtip **Comparable<E>**.

Drugi primer, u sledećem kodu (programu), vi deklarišete sirovi tip **stack** u liniji 1, dodeljujete mu **new GenericStack<String>** u liniji 2, i gurate string i objekat celog broja (integer) u **stack** u linijama 3 i 4

```
GenericStack stack;  
stack = new GenericStack<String>();  
stack.push("Welcome to Java");  
stack.push(new Integer(2));
```

Međutim, linja 4 nije bezbedna jer je stek planiran da memoriše stringove, a u njega je smešten objekat celog broja (Integer). Linja 3 je u redu, međutim, kompajler će ukazati na greške u linijama 3 i 4 jer ne može da sledi semantičko značenje programa. Sve što kompajler

zna je da je stek sirovi tip i da izvršava nebezbedno neke operacije. Prema tome, prikazuje se upozorenje na buduće greške.

▼ Poglavlje 7

Džoker generički tipovi

ŠTA JE DŽOKER?

Možete koristiti neograničene džokere, ograničene džokere ili nisko-ograničene džokere da bi specificirali opseg nekog generičkog tipa

Šta je džoker generički tip i zašto je on potreban? Da bi pokazali potrebu za džokerima, analiziramo primer na slici 1. Primer definiše generički metod **max()** koji nalazi maksimalnu vrednost brojeva smeštenih u steku (stack). Metod **main** kreira stek objekata celih brojeva (**integer objects**) dodaje tri cela broj u stek i poziva metod **max()** radi nalaženja maksimalne vrednosti brojeva u steku.

```
public class WildCardNeedDemo {
    public static void main(String[] args ) {
        GenericStack<Integer> intStack = new GenericStack<>();
        intStack.push(1); // 1 je autboksovan new Integer(1)
        intStack.push(2);
        intStack.push(-2);

        System.out.print("The max number is " + max(intStack));
    }

    /** Nalaženje maksimuma u steku brojeva */
    public static double max(GenericStack<Number> stack) {
        double max = stack.pop().doubleValue(); // Inicijalizacija max

        while (!stack.isEmpty()) {
            double value = stack.pop().doubleValue();
            if (value > max)
                max = value;
        }

        return max;
    }
}
```

Program na slici 1 ima grešku kompilacije u liniji 8 jer **intStack** nije objekt klase **GenericStack<Number>**. Zato, ne možemo da pozovemo metod **max(intStack)**

Činjenica je da Integer pod-tip od **Number**, ali **GenericStack<Integer>** nije pod-tip od **GenericStack<Number>**. Da bi se rešio ovaj problem, upotrebićemo džokera generičkih

tipova. Džoker generičkih tipova ima tri forme: `?` i `? extend T`, kao i `? super T`, gde je `T` generički tip.

Prva forma, `?`, naziva se neograničeni džoker (engl., *unbounded wildcard*), isti je kao i `? extend Object`. Druga forma, `? extends T`, naziva se *ograničeni džoker*, predstavljen sa `T` ili sa nepoznatim pod-tipom `T`. Treća forma, `? super T`, se naziva džoker sa donjom granicom, označen sa `T` ili sa nepoznatim pod-tipom `T`. Otklanjanje greške se vrši zamenom linje 12 u listingu klase `WildcardNeedDemo` na sledeći način:

```
public static double max(GenericStack<? extends Number> stack) {
```

`<? extends Number>` je džoker tipa koji predstavlja `Number` ili pod-tip od `Number`, te je ispravno da se pozove metod `max(new GenericStack<Integer>())` ili `max(new GenericStack<Double>())`.

PRIMENA DŽOKER TIPRA

Džoker `<?>` predstavlja bilo koji tip objekta.

Listing klase `AnyWildcardDemo` pokazuje primer korišćenja `?` džokera u metodi `print`, koji štampa objekte u steku i prazni stek. `<?>` je džoker koji predstavlja bilo koji tip objekata. On je ekvivalentan sa `<? extends Object>`. Šta se dešava kada vi zamenite `GenericStack<?>` sa `GenericStack<Integer>?` Bilo bi pogrešno da se pozove `print(intStack)`, jer `intStack` nije objekat klase `GenericStack<Object>`. Takođe, `GenericStack<Integer>` nije podtip `GenericStack<Object>`, iako je `Integer` pod-tip od `Object`.

```
public class AnyWildcardDemo {
    public static void main(String[] args ) {
        GenericStack<Integer> intStack = new GenericStack<Integer>();
        intStack.push(1); // 1 se transformiše u new Integer(1)
        intStack.push(2);
        intStack.push(-2);

        print(intStack);
    }

    /** Štampa objekte i prazni stek */
    public static void print(GenericStack<?> stack) {
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }
}
```

Kada je džoker `<? super T>` potreban? Pogledajmo primer na klase `SuperWildcardDemo`. Primer kreira stek stringova `stack1` (linija 3) i stek objekata `stack2` (linija 4) i poziva metod `add(stack1, stack2)` (linija 8) kojom dodaje stringove u `stack1` i `stack2`. `GenericStack<? super T>` je upotrebljen radi deklaracije `stack2` u liniji 13. Ako se `<? super T>` zameni

sa **<T>**, javiće se greška kompilacije kod **add(stack1, stack2)** u liniji 8, jer je tip steka **stack1 GenericStack<String>**, i steka **stack2** je **GenericStack<Object>**. **<? super T>** predstavlja tip **T** ili pod-tip od **T**. **Object** je supertip od **String**.

```
public class SuperWildcardDemo {
    public static void main(String[] args) {
        GenericStack<String> stack1 = new GenericStack<String>();
        GenericStack<Object> stack2 = new GenericStack<Object>();
        stack2.push("Java");
        stack2.push(2);
        stack1.push("Sun");
        add(stack1, stack2);
        AnyWildcardDemo.print(stack2);
    }

    public static <T> void add(GenericStack<T> stack1,
        GenericStack<? super T> stack2) {
        while (!stack1.isEmpty())
            stack2.push(stack1.pop());
    }
}
```

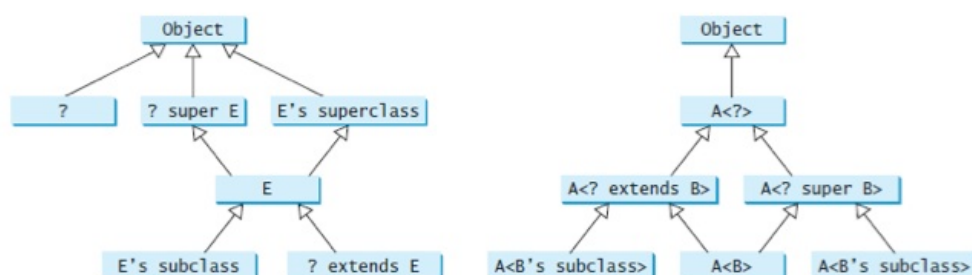
VEZE NASLEĐIVANJA GENERIČKIH TIPOVA I DŽOKER TIPOVA

Džoker tip, kao i generičke klase, se mogu nasleđivati.

Programski kod klase **SuperWildcardDemo** će takođe raditi ako se zaglavlje metoda u linijama 12 i 13 zameni sa:

```
public static <T> void add(GenericStack<? extends T> stack1,
    GenericStack<T> stack2)
```

Veze nasleđivanja generičkih tipova i džoker tipova su prikazane na slici 1. Na toj slici, **A** i **B** predstavljaju klase ili interfejsa, a **E** je parametar generičkog tipa.



Slika 7.1 Veze između generičkih tipova i džoker tipova [1]

▼ Poglavlje 8

Brisanje tipova

ŠTA JE BRISANJE TIPA?

Informacija o genericima koji se upotrebljavaju za vreme kompilacije, a koji nisu raspoloživi u vreme izvršenja. To se zove brisanje tipa.

Generici se primenjuju upotrebom jednog pristupa koji se naziva brisanje tipa: kompajler upotrebljava informaciju o generičkom tipu da bi kompilirao kod, ali ga posle briše. Zbog toga, informacija o genericima nije raspoloživa u vreme izvršenja koda. Ovaj pristup omogućava generički kod da ostvari kompatibilnost unazad sa starim kodom koji upotrebljava sirove tipove.

Generici su prisutni samo u vreme kompilacije. Kada kompajler potvrdi da je neki generički tip upotrebljen bezbedno, on ga konvertuje u sirovi tip. Na primer, kompajler proverava da li kod na slici 1a na korektan način koristi generike i onda ga prevodi u ekvivalentan kod dat na slici 1b u vreme izvršavanja koda. Kod na slici 1b koristi sirovi tip.

<pre>ArrayList<String> list = new ArrayList<String> (); list.add("Oklahoma"); String state = list.get(0);</pre>	<pre>ArrayList list = new ArrayList (); list.add("Oklahoma"); String state = (String)list.get(0);</pre>
(a)	(b)

Slika 8.1 Prevođenje generika u sirove tipove od strane kompajlera [1]

Kada se kompiliraju generičke klase, interfejsi i metodi, kompajler zamenjuje generički tip sa tipom Object. Na primer, kompajler bi preveo metod dat na slici 2a u metod na slici 2b

<pre>public static <E> void print(E[] list) { for (int i = 0; i < list.length; i++) System.out.print(list[i] + " "); System.out.println(); }</pre>	<pre>public static void print(Object [] list) { for (int i = 0; i < list.length; i++) System.out.print(list[i] + " "); System.out.println(); }</pre>
(a)	(b)

Slika 8.2 Prevođenje metoda print sa generikom u metod sa sirovim tipom od strane kompajlera [1]

Ako je generički tip ograničen, kompajler ga zamenjuje sa ograničenim tipom. Na primer, kompajler će metod na slici 3a, zameniti s metodom na slici 3b.

<pre>public static <E extends GeometricObject> boolean equalArea(E object1, E object2) { return object1.getArea() == object2.getArea(); }</pre>	<pre>public static boolean equalArea(GeometricObject object1, GeometricObject object2) { return object1.getArea() == object2.getArea(); }</pre>
(a)	(b)

Slika 8.3 Prevođenje metoda sa generikom u metod sa sirovim tipom od strane kompajlera [1]

Važna napomena je da generička klasa se deli sa svim objektima bez obzira od njihovog stvarnog i konkretnog tipa.

PRIMER SA KLASOM ARRAYLIST I OGRANIČENJA

Generički tipovi se brišu u vreme kompilacije.

Predpostavimo da su list1 i list2 kreiranje na sledeći način:

```
ArrayList<String> list1 = new ArrayList<>();  
ArrayList<Integer> list2 = new ArrayList<>();
```

Iako su **ArrayList<String>** i **ArrayList<Integer>** dva tipa u vreme kompilacije, samo klasa **ArrayList** se unosi u JVM u vreme izvršenja. **list1** i **list2** su objekti klase **ArrayList**, tako da su sledeća dva iskaza istinita (true):

```
System.out.println(list1 instanceof ArrayList);  
System.out.println(list2 instanceof ArrayList);
```

Međutim, iskazi **list1 instanceof ArrayList<String>** je pogrešan. Kako se **ArrayList<String>** ne smešta kao posebna klasa u JVM, upotreba u vreme izvršenja nema nikakvog smisla.

Kako se generički tipovi brišu u vreme izvršenje programa, postoje određena ograničenja kako se generički tipovi mogu da koriste. Navećemo neka od ograničenja:

BRISANJE GENERIČKIH TIPOVA BEZ OGRANIČENJA

Kako tip parametra T nije ograničen, Java kompajler ga zamenjuje sa tipom Object

Za vreme procesa brisanja tipova, Java kompajler briše sve tipove parametara i svaki od njih zamenjuje sa njegovim prvim ograničenjem ako je tip parametra ograničen, ili zamenjuje sa tipom **Object** ako tip parametra nije ograničen.

Pogledajmo sledeću generičku klasu koja predstavlja čvor u povezanoj listi.

```
public class Node<T> {  
  
    private T data;  
    private Node<T> next;
```

```
public Node(T data, Node<T> next) {
    this.data = data;
    this.next = next;
}

public T getData() { return data; }
// ...
}
```

Kako tip parametra **T** nije ograničen, Java kompajler ga zamenjuje sa tipom **Object**

```
public class Node<T> extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

BRISANJE GENERIČKIH TIPOVA SA OGRANIČENJEM

Java kompajler zamenjuje ograničen tip parametra T sa prvom povezanom klasom, a to je klasa Comparable

U sledećem primeru, generička klasa **Node** upotrebljava ograničen tip parametra.

```
public class Node<T> extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Java kompajler zamenjuje ograničen tip parametra **T** sa prvom povezanom klasom, a to je klasa **Comparable**.

```
public class Node {  
  
    private Comparable data;  
    private Node next;  
  
    public Node(Comparable data, Node next) {  
        this.data = data;  
        this.next = next;  
    }  
  
    public Comparable getData() { return data; }  
    // ...  
}
```

BRISANJE GENERIČKIH METODA

Java kompajler briše tipove parametara u argumentima generičkih metoda

Java kompajler takođe briše tipove parametara u argumentima generičkih metoda. Pogledajmo sledeći generički metod:

```
// Broji broj javljanja elemenata u anArray .  
//  
public static <T> int count(T[] anArray, T elem) {  
    int cnt = 0;  
    for (T e : anArray)  
        if (e.equals(elem))  
            ++cnt;  
    return cnt;  
}
```

Kako **T** nije ograničen, Java kompajler ga zamenjuje sa **Object**.

```
public static int count(Object[] anArray, Object elem) {  
    int cnt = 0;  
    for (Object e : anArray)  
        if (e.equals(elem))  
            ++cnt;  
    return cnt;  
}
```

Pretpostavimo da su definisane sledeće klase:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

Možete napisati generički metod za crtanje različitih geometrijskih oblika:

```
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

Java kompajler zamenjuje **T** sa **Shape**.

```
public static void draw(Shape shape) { /* ... */ }
```

EFEKTI BRISANJA TIPOVA

Brisanje tipa proizvodi neočekivane efekte

Ponekad, brisanje tipa proizvodi neočekivane efekte. Sledeći primer pokazuje kako se mogu ti neočekivani efekti da jave. Primer pokazuje kako kompajler kreira sintetički metod, koji se naziva **metod mosta** (*bridge method*), kao deo procesa brisanja tipa.

Date su sledeće dve klase:

```
public class Node<T> {

    public T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node<Integer> {
    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

Pogledajmo sledeći programski kod:

```
MyNode mn = new MyNode(5);
Node n = mn; // Sirovi tip - kompajler iybacuje upozorenje neproveravanja
n.setData("Hello");
```

```
Integer x = mn.data; // Prouzrukuje izbacivanje izuzetka ClassCastException
```

Posle brisanja tipa, kod postaje:

```
yNode mn = new MyNode(5);
Node n = (MyNode)mn; // Sirovi tip - kompajler izbacuje upozorenje
neproveravanja
n.setData("Hello");
Integer x = (String)mn.data; // Prouzrukuje izbacivanje izuzetka
ClassCastException
```

Evo šta se dešava kada se ovaj program izvršava:

1. **n.setData("Hello");** - prouzrukuje izvršenje metoda **setData(Object)** nad objektom klase **MyNode** (Klasa **MyNode** nasleđuje metod **setData(Object)** iz klase **Node**.)
2. U telu metoda **setData(Object)**, atribut **n** se dodeljuje **String**.
3. atribut istog objekta označen sa **mn**, postaje ceo broj (jer **mn** je **MyNode** je podklasa od **Node<Integer>**).
4. Pokušaj da se **String** dodeli tipu **Integer** prouzrukuje javljanje **ClassCastException** kao rezultat konverzije r.

METODI MOSTA

Prilikom kompilacije neke klase ili interfejsa koji proširuje parametizovanu klasu ili primenjuje parametizovani interfejs, kompajler može da kreira sintetički metod koji se naziva metod mosta

Prilikom kompilacije neke klase ili interfejsa koji proširuje parametrizovanu klasu ili primenjuje parametrizovani interfejs, kompajler može da kreira sintetički metod koji se naziva metod mosta, kao deo procesa brisanja tipa. Vi ne morate da se oko toga brinete, ali se možete zbuniti kada se se metod mosta pojavi u steku.

Posle brisanja tipa, klase **Node** i **MyNode** postaju:.

```
public class Node {

    public Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
```

```
public class MyNode extends Node {  
  
    public MyNode(Integer data) { super(data); }  
  
    public void setData(Integer data) {  
        System.out.println("MyNode.setData");  
        super.setData(data);  
    }  
}
```

Posle brisanja tipa, ne slaže se potpisi metoda. Metod **Node** postaje **setData(Object)** i metod **MyNode** postaje **setData(Integer)**. Prema tome, metod **MyNode setData** ne redefiniše metod **Node setData**.

Da bi se ovaj problem rešio i zadržao polimorfizam generičkih tipova posle brisanja tipa, Java kompajler generiše metod mosta radi obezbeđivanja da se radi sa podtippovima, kao što se očekuje. Za klasu **MyNode**, kompajler generiše sledeći metod mosta za **setData**:

```
class MyNode extends Node {  
  
    // Metod mosta koji generiše kompajler  
    //  
    public void setData(Object data) {  
        setData((Integer) data);  
    }  
  
    public void setData(Integer data) {  
        System.out.println("MyNode.setData");  
        super.setData(data);  
    }  
  
    // ...  
}
```

Metod mosta ima isti potpis kao metod **setData** klase **Node**, posle brisanja tipa delegira originalni **setMetod** metod.

▼ Poglavlje 9

Ograničenja primene generičkih tipova

OGRANIČENJA U PRIMENI GENERIČKIH TIPOVA

Upotrebu generičkih tipova u Javi prati i određena ograničenja o kojima morate da vodite računa:

Upotrebu generičkih tipova u Javi prati i određena ograničenja o kojima morate da vodite računa:

1. Ne može da se kreira istanca (objekat) upotrebom parametra generičkog tipa
2. Ne možete kreirati niz (array) upotrebom parametra generičkog tipa
3. Parametar generičkog tipa neke klase nije dozvoljen u statičkom kontekstu
4. Klase izuzetka ne mogu da budu generičke, tj. ne mogu se kreirati, hvatati (**catch**) ili izbacivati (**throw**) objekti sa parametrizovanim tipovima
5. Ne možete primeniti generičke tipove sa primitivnim tipovima
6. Ne možete upotrebiti konverziju tipova ili koristiti objekte sa parametrijovanim tipovima
7. Ne možete preopteretiti (**overload**) metod sa formalnim tipovima parametara svakog brisanja preopterećenja u isti sirovi tip.

Mi ćemo ovde detaljnije objasniti četiri od ovih ograničenja, a za ostale vam preporučujemo korišćenje informacije na: <https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>

OGRANIČENJA PRIMENE GENERIČKIH TIPOVA 1 I 2

Postoje određena ograničenja kako se generički tipovi mogu da koriste. Ne mogu se upotrebljavati E() i E[].

1. **Ne može da se upotrebljava new E():** Ne može da se kreira istanca (objekat) upotrebom parametra generičkog tipa. Na primer, sledeći iskaz je pogrešan

```
E object = new E();
```

Razlog za ove je u činjenici da se **newE()** izvršava za vreme rada programa, a tada generički tip **E** nije raspoloživ (**generički tipovi su raspoloživi samo za vreme kompilacije**).

2. **Ne može da se upotrebljava new E[]:** Ne možete kreirati niz (**array**) upotrebom parametra generičkog tipa. Na primer, sledeći iskaz je pogrešan


```
E[] elements = (E[])new Object[capacity];
```

Možete zaobići ovo ograničenje kreiranjem niza tipa **Object** i onda konvertovati ga u tip **E[]**:

```
[] elements = (E[])new Object[capacity];
```

Međutim, moguće je dobiti upozoravajuću poruku prilikom kompilacije. Razlog za to je što kompajler nije siguran da će konverzija uspeti u fazi rada programa. Na primer, ako je **E** tipa **String** i **new Object []** je niz objekata tipa **Integer**, onda će **(String[])(new Object[])** poruzorokovati javljanje **ClassCastException**. Ova vrsta upozorenja u vreme kompilacije je ograničenje Java generika i ne može se izbeći.

Kreiranje generičkog niza upotrebom neke generičke klase nije dozvoljeno, takođe. Na primer, sledeći kod je pogrešan

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

Vi možete da upotrebite sledeći kod da bi se rešili ovog ograničenja:

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

Međutim, ovde ćete dobiti upozorenje prilikom kompilacije.

OGRANIČENJA PRIMENE GENERIČKIH TIPOVA 3 I 4

Parametar generičkog tipa neke klase nije dozvoljen u statičkom kontekstu. Klase izuzetka ne mogu da budu generičke.

Kako sve instance (objekti) generičke klase imaju istu klasu u vreme izvršenja, statičke promenljive i metodi generičke klase se dela od strane svih instanci. Prema tome, nepravilno je pozvati parametar generičkog tipa klase u nekom statičkom metodu, polju ili u inicijalizatoru.

```
public class Test<E> {  
    public static void m(E o1) { // Nepravilno  
    }  
    public static E o1; // Nepravilno  
    static {  
        E o2; // Nepravilno  
    }  
}
```

Generička klasa ne mogu da prošire **java.lang.Throwable**. Sledeći iskaz bi bio nepravilan:

```
public class MyException<T> extends Exception {  
}
```

Zašto? Kada bi bilo dozvoljno, mogli bi da uhvatiti (catch) klauzulu za **MyException<E>** na sledeći način:

```
try {  
    ...  
}  
catch (MyException<T> ex) {  
    ...  
}
```

JVM treba da proveri izbačen izuzetak iz **try** klauzule da bi proverila da li odgovara tipu koji je specificiran u **catch** klauzuli. To je nemoguće, jer ta vrsta informacije nije raspoloživa u vreme izvršenja programa.

▼ Poglavlje 10

Vežba – Pokazni primeri

PRIMER 1

Cilj ovog primera je provežbavanje generičkih klasa sa jednim parametrom

Napraviti klasu ProstGenerik koja predstavlja generičku klasu koja prima kao parametar bilo koji tip parametra (T). Unutar klase napraviti metodu printType koja vraća ime klase prosleđenog objekta

Potrebno vreme: 7 minuta

Klasa ProstGeneric:

```
class ProstGeneric<T>{  
  
    private T objReff = null;  
  
    public ProstGeneric(T param){  
        this.objReff = param;  
    }  
  
    public T getObjReff(){  
        return this.objReff;  
    }  
  
    public void printType(){  
        System.out.println("Type: "+objReff.getClass().getName());  
    }  
}
```

Klasa Main:

```
public class Main {  
  
    public static void main(String[] args) {  
        new Main();  
    }  
  
    public Main() {  
        ProstGeneric<String> sgs = new ProstGeneric<String>("JAVA2NOVICE");  
        sgs.printType();  
        ProstGeneric<Boolean> sgb = new ProstGeneric<Boolean>(Boolean.TRUE);  
    }  
}
```

```
        sgb.printType();  
    }  
}
```

PRIMER 1 - OBJAŠNJENJE

Objašnjenje programskog koda primera 1

U ovom zadatku kreiramo klasu koja ima jedan atribut generičkog tipa T.

Klasa **ProstGenerik** sadrži konstruktor, getter za generički atribut i metodu koja štampa tip generičkog atributa koji je naveden. Štampanje tipa podrazumeva štampanje naziva konkretne klase koja će biti definisana i može se dobiti tako što prvo pozovemo metodu **getClass()** koja sadrži određene informacije o samoj referenci uključujući i naziv klase. Konkretna naziv klase se dobija pozivom metode getName().

Svaki put kada kreiramo novu instancu tipa ProstGenerik u dijamatskim zagradama (<>) moramo definisati tip odnosno klasu koja se odnosi na generički tip.

PRIMER 2

Cilj ovog primera je provežbavanje generičkih klasa sa dva parametra

Napraviti klasu GenerikSaDvaParametra koja prima dva objekta i metodu printType koja vraća ime klase oba prosleđena objekta.

Potrebno vreme: 7 minuta

Klasa GenerikSaDvaParametra:

```
public class GenerikSaDvaParametra<U, V>{  
  
    private U objUreff;  
    private V objVreff;  
  
    public GenerikSaDvaParametra(U objU, V objV){  
        this.objUreff = objU;  
        this.objVreff = objV;  
    }  
  
    public void printTypes(){  
        System.out.println("U Type: "+this.objUreff.getClass().getName());  
        System.out.println("V Type: "+this.objVreff.getClass().getName());  
    }  
}
```

Klasa Main:

```
public class Main {  
    public static void main(String[] args) {  
        new Main();  
    }  
  
    public Main(){  
        GenerikSaDvaParametra<String, Integer> sample  
            = new GenerikSaDvaParametra<String, Integer>("CS202 JE SUPER",  
100);  
        sample.printTypes();  
    }  
}
```

PRIMER 2 - OBJAŠNJENJE

Objašnjenje programskog koda primera 2

Ovaj primer demonstrira rad sa klasom koja sadrži dva atributa generičkog tipa. Za generičke tipove smo definisali vrednosti U i V s tim što je za te vrednosti moguće koristiti i bilo koje drugo slovo. Važno je naglasiti da definisani generički atributi mogu biti i istog i različitog tipa u zavisnosti od potreba programa.

Klasa GenerikSaDvaParametra pored konstruktora sadrži i metodu printTypes koja štampa definisane tipove odnosno nazive konkretnih klasa generičkih atributa što se realizuje identično kao u prethodnom zadatku.

Sada svaki put kada kreiramo instancu klase GenerikSaDvaParametra potrebno je da definišemo tip podatka za oba generička atributa kao što je urađeno u klasi Main.

PRIMER 3

Cilj ovog primera je provežbavanje generičkih klasa sa parametrom koji nasleđuje klasu

Napraviti klasu A koja ima metodu printClass koja ispisuje da je ona super klasa A. Potom napraviti klase B i C koje nasleđuju klasu A i imaju svoje implementacije metode printClass koje vraćaju da su pod klase klase A. Napraviti klasu GenericiNasledjivanje koja prima parametar T koji nasleđuje klasu A. Napraviti u ovoj klasi metodu doRunTest koja zove printClass nad prosleđenom T objektu koji nasleđuje A.

Potrebno vreme: 7 minuta

Klasa GenericiNasledjivanje:

```
public class GenericiNasledjivanje<T extends A> {  
  
    private T objRef;
```

```
public GenericiNasledjivanje(T obj) {
    this.objRef = obj;
}

public void doRunTest() {
    this.objRef.printClass();
}
}

class A {

    public void printClass() {
        System.out.println("I am in super class A");
    }
}

class B extends A {

    public void printClass() {
        System.out.println("I am in sub class B");
    }
}

class C extends A {

    public void printClass() {
        System.out.println("I am in sub class C");
    }
}
```

Klasa Main:

```
public class Main {

    public static void main(String[] args) {
        new Main();
    }

    public Main() {
        GenericiNasledjivanje<C> bec = new GenericiNasledjivanje<C>(new C());
        bec.doRunTest();
        GenericiNasledjivanje<B> beb = new GenericiNasledjivanje<B>(new B());
        beb.doRunTest();
        GenericiNasledjivanje<A> bea = new GenericiNasledjivanje<A>(new A());
        bea.doRunTest();
    }
}
```

PRIMER 3 - OBJAŠNJENJE

Objašnjenje programskog koda primera 3

U ovom zadatku je napravljena hijerarhija klasa, gde je klasa A nadklasa ili superklasa, dok su klase B i C podklase odnosno klase koje nasleđuju klasu A.

U okviru klase **GenericiNasledjivanje** definišemo generički tip koristeći ograničavanje generika sa gornje strane. Dakle, kada definišemo generik korišćenjem klauzulu extends u formatu <T extends A> to podrazumeva da taj generički atribut mora pripadati ili klasi A ili klasi koja nasleđuje klasu A, čime se ograničava mogućnost definisanja tipova.

PRIMER 4

Cilj ovog primera je provežbavanje generičkih klasa sa parametrom koji implementira interfejs

Napraviti interfejs D koja ima metodu printClass. Potom napraviti klase G i F koje implementiraju klasu D i imaju svoje implementacije metode printClass koje vraćaju da su implementacije klase D. Napraviti klasu GenericiInterfejsi koja prima parametar T koji nasleđuje interfejs D. Napraviti u ovoj klasi metodu doRunTest koja zove printClass nad prosleđenom T objektu koji implementira D.

Potrebno vreme: 5 minuta

Klasa GenericiInterfejsi:

```
class GenericiInterfejsi<T extends D>{

    private T objRef;

    public GenericiInterfejsi(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.printClass();
    }
}

interface D{
    public void printClass();
}

class G implements D{
    public void printClass(){
        System.out.println("I am in class D");
    }
}

class F implements D{
    public void printClass(){
        System.out.println("I am in class Z");
    }
}
```

```
}
}
```

Klasa Main:

```
public class Main {
    public static void main(String[] args) {
        new Main();
    }
    public Main(){
        GenericiInterfejsi<G> bey = new GenericiInterfejsi<G>(new G());
        bey.doRunTest();
        GenericiInterfejsi<F> bez = new GenericiInterfejsi<F>(new F());
        bez.doRunTest();
    }
}
```

U klasi Generici Interfejsi je takođe demonstrirano ograničavanje generika sa gornje strane. U ovom primeru možemo zaključiti da se klauzulom extends može ograničiti tip generika u zavisnosti od interfejsa koji klase implementiraju.

Pošto je D interfejs, definisanjem generika u formatu <T extends D> ograničavamo tip generika na klase koje implementiraju interfejs D. Pošto klase F i G implementiraju D, one predstavljaju validan tip generika klase Generici Interfejsi, dok bi u protivnom u fazi kompajliranja bila prijavljena greška.

PRIMER 5

Cilj ovog primera je provežbavanje generičkih klasa sa univerzalnim metodama za upoređivanje klasa koji poštuju istu nad klasu

Napraviti klasu Zaposleni koja ima od podataka ime i platu. Napraviti klase CompAZap i CompBZap koje nasleđuju klasu Zaposleni. Napraviti klasu MyEmployeeUtil koja ima generički parametar koji treba da bude zaposleni. Ima metodu za opšte uzimanje plate kao i metodu koja proverava da li je plata ista kao nekom drugom zaposlenom.

Potrebno vreme: 7 minuta

Klasa MyEmployeeUtil:

```
class MyEmployeeUtil<T extends Zaposleni>{

    private T zap;

    public MyEmployeeUtil(T obj){
        zap = obj;
    }

    public int getPlata(){
        return zap.getPlata();
    }
}
```



```
    }

    public boolean isSalaryEqual(MyEmployeeUtil<?> otherEmp){

        if(zap.getPlata()== otherEmp.getPlata()){
            return true;
        }
        return false;
    }
}

class Zaposleni{

    private String ime;
    private int plata;

    public Zaposleni(String ime, int plata){
        this.ime = ime;
        this.plata = plata;
    }

    public String getIme() {
        return ime;
    }
    public void setIme(String ime) {
        this.ime = ime;
    }
    public int getPlata() {
        return plata;
    }
    public void setPlata(int plata) {
        this.plata = plata;
    }
}

class CompAZap extends Zaposleni{

    public CompAZap(String nm, int sal){
        super(nm, sal);
    }
}

class CompBZap extends Zaposleni{

    public CompBZap(String nm, int sal){
        super(nm, sal);
    }
}
```

Klasa MainClass:

```
public class MainClass {
```

```
public static void main(String a[]){

    MyEmployeeUtil<CompAZap> empA
        = new MyEmployeeUtil<CompAZap>(new CompAZap("Ram", 20000));
    MyEmployeeUtil<CompBZap> empB
        = new MyEmployeeUtil<CompBZap>(new CompBZap("Krish", 30000));
    MyEmployeeUtil<CompAZap> empC
        = new MyEmployeeUtil<CompAZap>(new CompAZap("Nagesh", 20000));
    System.out.println("Is salary same? "+empA.isSalaryEqual(empB));
    System.out.println("Is salary same? "+empA.isSalaryEqual(empC));

}
}
```

PRIMER 5 - OBJAŠNJENJE

Objašnjenje programskog koda primera 5

I u ovom zadatku pravimo odgovarajuću hijerarhiju klasa gde je POJO (Plain Old Java Object) klasa Zaposleni superklasa dok su klase CompAZap i CompBZap podklase koje je nasleđuju.

U klasi MyEmployeeUtil imamo generički tip koji se ograničava na klase tipa Zaposleni ili klase koje je nasleđuju. Zbog tog ograničenja je moguće da preko metode isSalaryEqual poredimo plate zaposlenih koji pripadaju bilo kojoj klasi u definisanoj hijerarhiji (Zaposleni ili njene podklase).

Ograničenjem generika garantovano imamo metodu getPlata koja je definisana u graničnoj superklasi Zaposleni preko koje pristupamo vrednostima plata kako bi smo izvršili poređenje. Na ovaj način svaka podklasa može da primeni drugačiju formulu za izračunavanje plata ili može override-ovati metodu getPlata u podklasama s tim što je sama funkcionalnost poređenja plata apstrahovana u okviru klase MyEmployeeUtil.

PRIMER 6

Cilj ovog primera je da se provežba rad sa generičkim metodama

Kreirati klasu GenericsUtility koja ima sledeće statičke metode:

- generičku metodu koja prima ArrayList-u i uklanja sve duplikate iz ArrayList-e
- generičku metodu koja prima dvodimenzionalni niz i pronalazi maksimalan element u njemu
- generičku metodu koja pravi nasumičan raspored elemenata u ArrayList-i koju prima kao parametar
- generičku metodu koja pronalazi indeks elementa koji se prosleđuje kao parametar i vraća -1 ukoliko element ne postoji u nizu; niz se takođe prosleđuje kao parametar

Takođe napraviti Main klasu u kojoj ćete testirati rad statičkih metoda

Potrebno vreme: 7 minuta

PRIMER 6 - REŠENJE

Programski kod za primer 6

Klasa MyEmployeeUtil:

```
public class GenericsUtility {

    // ukljanja duplikate iz ArrayList-e
    public static <E> ArrayList<E> removeDuplicates(ArrayList<E> list) {

        for (int i = 0; i < list.size(); i++) {
            for (int j = i + 1; j < list.size(); j++) {
                if(list.get(i).equals(list.get(j))){
                    list.remove(j);
                }
            }
        }
        return list;
    }

    // pronalazi maksimalni element matrice
    public static <E extends Comparable<E>> E max(E[][] matrix){
        E max = matrix[0][0];
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[0].length; j++) {
                if(matrix[i][j].compareTo(max) > 0){
                    max = matrix[i][j];
                }
            }
        }
        return max;
    }

    // pravi nasumican raspored elemenata u listi
    public static <E> void shuffle(ArrayList<E> list){
        Random rand = new Random();
        for (int i = 0; i < list.size(); i++) {
            int randIndex = rand.nextInt(list.size());
            E pom = list.get(i);
            list.set(i, list.get(randIndex));
            list.set(randIndex, pom);
        }
    }

    // pretrazuje element niza i vraca vrednost indeksa ukoliko postoji i -1
    ukoliko element ne postoji
    public static <E extends Comparable<E>> int indexSearch(E[] arr, E val){
        int index = -1;
```

```

        for (int i = 0; i < arr.length; i++) {
            if(arr[i].equals(val)){
                index = i;
            }
        }
        return index;
    }
}

```

Main klasa:

```

public class GenericiMain {

    private Integer[] arr = new Integer[]{1, 3, 5, 0, 3, 5, 6, 10};
    private ArrayList<Integer> list = new ArrayList<>(Arrays.asList(arr));
    private String[][] niz2D = new String[][]{{"Rados", "Pavlicevic"}, {"Jovan",
"Antonic"}, {"Stevan", "Rakic"},
{"Petar", "Ciric"}, {"Milena", "Vuckovic"}}};

    public GenericiMain() {

        System.out.println("ArrayList: ");
        for(Integer tmp : list){
            System.out.println(tmp);
        }
        GenericsUtility.shuffle(list);
        System.out.println("Shuffled ArrayList: ");
        for(Integer tmp : list){
            System.out.print(tmp + " ");
        }
        System.out.println("\nArrayList with duplicates removed: ");
        list = GenericsUtility.removeDuplicates(list);
        for(Integer tmp : list){
            System.out.println(tmp);
        }

        System.out.println("Max element of 2dArray: ");
        System.out.println(GenericsUtility.max(niz2D));
        System.out.println("");
        System.out.println(GenericsUtility.indexSearch(arr, 10));

    }

    public static void main(String[] args) {
        new GenericiMain();
    }
}

```

PRIMER 6 - OBJAŠNJENJE

Objašnjenje programskog koda primera 6

U ovom primeru potrebno je kreirati generičke metode kako bi se funkcionalnosti koje te metode obezbeđuju generalizovale odnosno kako bi bile nezavisne od tipa podataka. Ove metode su objedinjene u okviru klase `GenericsUtility` koja obezbeđuje globalni pristup tim metodama tako što su definisane kao `public static`.

Kod metoda `shuffle` i `removeDuplicates` imamo neograničene generike što znači da se funkcionalnosti koje te metode pružaju mogu primeniti na svim tipovima koje nasleđuju klasu `Object` (metoda `removeDuplicates` koristi metodu `equals` koja se nasleđuje iz klase `Object`). Pored toga imamo metode `max` i `indexSearch` koje koriste metodu `compareTo` koja pripada interfejsu `Comparable` pa je zbog toga neophodno da se generički tip u okviru ovih metoda ograniči na klase koje implementiraju interfejs `Comparable` kako bi mogle da realizuju namenjene funkcionalnosti.

Metoda `removeDuplicates` prima `ArrayList`-u kao parametar i iz nje uklanja sve duplikate tako što proverava kroz dve ugnježdene petlje poredi svaki element liste sa svakim i kada naiđe na poklapanje vrednosti vrši izbacivanje jednog od elemenata iz liste.

Metoda `max` prima matricu kao parametar i izračunava maksimalni element matrice. Prvi element matrice se proglasi za trenutni maksimum, a zatim se prolazi kroz sve elemente matrice gde se proverava da li je neki element veći od trenutnog maksimuma i ukoliko jeste njegova vrednost se dodeli promenljivoj `max` koja se odnosi na trenutni maksimum.

Metoda `shuffle` vrši nasumičan raspored elemenata u listi koja joj se prosledi i to realizuje tako što za svaki element niza generiše nasumičan broj u opsegu od 0 do veličine liste i vrši zamenu mesta trenutnog elementa i elementa koji se nalazi na poziciji nasumično generisanog broja.

Metoda `indexSearch` prima listu i vrednost koja treba da se pretraži u listi i funkcioniše tako što prolazi kroz sve elemente liste dok ne naiđe na element koji je jednak traženoj vrednosti nakon čega vraća njegovu vrednost indeksa u listi (indeks prvog pojavljivanja vrednosti). Ukoliko se tražena vrednost ne nalazi u listi metoda vraća -1.

PRIMER 7

Cilj primera je da se provežba znanje o generičkim metodama (

Napisati generičku metodu koja prima dva niza i vrada listu koja sadrži elemente koji čine presek dva niza. Potrebno vreme: 5 minuta

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package zadatak.pkg7;

import com.sun.javafx.scene.control.skin.VirtualFlow;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author Jovana
 */
public class Zadatak7<T> {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        new Zadatak7();
    }

    public Zadatak7() {
        List<Integer> lista1 = new ArrayList<>();
        lista1.add(5);
        lista1.add(15);
        lista1.add(11);
        lista1.add(16);
        lista1.add(18);

        List<Integer> lista2 = new ArrayList<>();
        lista2.add(5);
        lista2.add(15);
        lista2.add(26);
        lista2.add(18);
        lista2.add(10);

        System.out.println(presek(lista1, lista2));

        List<String> lista3 = new ArrayList<>();
        lista3.add("a");
        lista3.add("abc");
        lista3.add("aab");
        lista3.add("bab");
        lista3.add("bba");

        List<String> lista4 = new ArrayList<>();
        lista4.add("aaa");
        lista4.add("abb");
        lista4.add("aab");
        lista4.add("bab");
        lista4.add("abba");

        System.out.println(presek(lista3, lista4));
    }
}
```

```
}

/**
 * @param lista1
 * @param lista2
 * @return
 */
public <T> List<T> presek(List<T> lista1, List<T> lista2) {
    List<T> presek = new ArrayList<>();

    for (T el : lista1) {
        if (lista2.contains(el)) {
            presek.add(el);
        }
    }
    return presek;
}
}
```

▼ Poglavlje 11

Vežba – Zadaci za individualni rad

ZADACI ZA SAMOSTALNI RAD - ZADACI OD 1 DO 5

Cilj zadatka je da student samostalno primeni korišćenje generičkih klasa

Zadatak 1 (15 minuta):

Napisati generičku metodu koja pronalazi indeks poslednjeg pojavljivanja nekog elementa u listi koju prima kao parametar.

Zadatak 2 (15 minuta):

Napisati generičku metodu koja vrši sortiranje elemenata niza. Kao parametre prima niz koji treba da se sortira kao i indikator. Indikator može biti 1 ili -1. Ukoliko je indikator 1 onda se niz treba sortirati u rastućem, a ukoliko je -1 u opadajućem poretку. Za sortiranje koristiti SelectionSort algoritam.

Zadatak 3 (15 minuta)::

Napisati generičku metodu koja pravi obrnut raspored elemenata u ArrayList-i koju prima kao parametar.

Zadatak 4 (15 minuta): :

Napisati generičku metodu koja proverava da li je niz simetričan. Niz je simetričan ako je raspored elemenata sa desna na levo identičan rasporedu elemenata sa leva na desno.

Zadatak 5 (15 minuta): :

Napisati generičku metodu koja iz liste uklanja sve elemente koji imaju veću vrednost od prosleđenog parametra. Za poređenje koristiti compareTo metodu interfejsa Comparable.

ZADACI ZA SAMOSTALNI RAD - ZADACI OD 6 DO 10

Cilj zadatka je da studenti samostalno primene znanja o generičkim metodama

Zadatak 6 (15 minuta): :

Napisati generičku metodu koja prima matricu i proverava da li su elementi na glavnoj dijagonali jednaki elementima na sporednoj dijagonali.

Zadatak 7 (15 minuta):

Kreirati sledeću hijerarhiju klasa. Klasa Zaposleni je super klasa i sadrži attribute ime, prezime i godine. Nakon toga kreirati pod klase Profesor i Asistent kojima treba dodati bar jedan dodatni atribut u odnosu na klasu zaposleni. Zatim kreirati JavaFX formu za dodavanje zaposlenih koja treba da sadrži inicijalna tri polja za unos imena, prezimena i godina. Pored toga potrebno je u formu dodati još RadioButton grupu ili ComboBox po izboru gde će se odabrati da li je zaposleni profesor ili asistent. Nakon odabira pomenute stavke potrebno je u formu dodati polja koja će se odnositi na dodatne attribute podklase. Nakon što se unesu svi parametri potrebno je dodati zaposlenog u ArrayList-u i u konzoli ispisati sve zaposlene iz liste.

Zadatak 8 (15 minuta):

Napisati generičku metodu koja prima dva niza i vraća listu koja sadrži elemente koji čine razliku dva niza.

Zadatak 9 (15 minuta):

Napisati generičku metodu koja prima varijabilan broj generičkih parametara i vraća njihovu aritmetičku sredinu.

Zadatak 10 (20 minuta): Ovo je neobaveyni yadatak ya rad kod kuće (ako se na vežbama ne stigne da uradi)

Napisati generičku metodu koja prima dva generička niza istog tipa i jedan integer; potrebno je sve elemente iz prvog niza koji se po indeksu pre prosleđenog integer-a prebaciti u drugi niz. Potrebno je prvo proširiti veličinu niza, a zatim dodati nove elemente

DOMAĆI ZADATAK

UPUTSTVO ZA SLANJE DOMAĆEG ZADATKA

Svaki student će od svog asistenta dobiti poseban domaći zadatak za lekciju CS202-L06.

Student je dužan da asistentu pošalje rešenje zadatka u roku od 7 dana od dana prijema zadatka.

Posle tog roka, rešenje zadatka se ocenjuje sa 50% umanjenja poena.

Prilikom slanja domaćeg zadatka svom asistentu neophodno je da ispunite sledeće:

Subject mail-a mora biti CS202-DZbr (u slučaju kada šaljete domaći za drugu nedelju to je CS202-DZ05)

U prilogu mail-a treba da se nalazi projekat koji se ocenjuje imenovan na sledeći način **CS202-DZbr-ImePrezimeBrojIndeksa**.

Na primer, CS202-DZ05-AleksandraArsic123

Poželjno je uraditi i printscreen koda pre pokretanja programa. Telo maila treba da ima pozdravnu poruku

Studenti prve grupe u Beogradu, kao i internet studenti zadatak na pregled šalju na mail lazar.mrkela@metropolitan.ac.rs

Studenti iz Niša zadatak na pregled šalju na mail jovana.jovic@metropolitan.ac.rs.

▼ Zaključak

REZIME

Pouke ove lekcije

- 10. Parametri generičkog tipa ne mogu se koristiti u klasama izuzetaka.
- 9. Ne možete upotrebiti parametar generičkog tipa klase u statičkom kontekstu.
- 8. Ne možete kreirati niz upotrebom parametra generičkog tipa.

- 1. Generici (generičke klase i metodi) vam daju mogućnost parametrizovanja tipova. Možete definisati klasu ili metod sa generičkim tipovima, koje kompajler zameni sa konkretnim tipovima.
- 2. Glavna korist od generika je što omogućavaju utvrđivanje grešaka prilikom kompilacije, umesto u fazi izvršenja programa.
- 3. Generička klasa ili metod dozvoljava vam da specificirate dozvoljene tipove objekata sa kojima klasa ili metod može da radi. Ako pokušate da upotrebljavate tu klasu ili metod sa drugim, nekompatibilnim objektom, kompajler će utvrditi grešku.
- 4. Generički tip koji je definisan u klasi, interfejsu ili u statičkom metodu, je formalni generički tip. On se kasnije može zameniti sa stvarnim i konkretnim tipom. Ta zamena generičkog tipa je generička konkretizacija (engl., generic instantiation).
- 5. Generička klasa, kao što je ArrayList koja se koristi bez parametra tipa, je sirov tip (engl., raw type). Upotreba sirovih tipova omogućava kompatibilnost unazad, tj. kompatibilnost sa ranijim verzijama Java.
- 6. Džoker generički tip ima tri forme: `?` i `? extends T`, i `? super T`, gde je `T` genrički tip.

- a. Prva forma `?`, je neograničeni džoker, je ista kao i `? extends Object`.
- b. Druga forma, `? extend T`, je ograničeni džoker, i predstavlja `T` ili nepoznati pod-tip `T`.
- c. Treća forma, `? super T`, je džoker sa ograničenom donjom granicom, označava `T` ili nepoznati pod-tip od `T`.
- 7. Ne možete kreirati instance upotrebom parametra generičkog tipa.

REFERENCE

Literatura primenjena u ovoj lekciji

- 1. Y. Daniel Liang, Introduction to Java Programming, Comprehensive Version, Chapter 19, 10th edition, Pierson – preporučeni udžbenik