Razvoj softvera

dr.sc. Emir Mešković

V predavanje

Sadržaj predavanja

- Generičke klase i metodi

Stack implementacija 1


```
class MojStack
{
   class Cvor
   {
     Cvor slijedeci_;
     Object vrijednost_;
     public Cvor(Object v, Cvor s)
     {
         slijedeci_ = s;
         vrijednost_ = v;
     }
   }
   private Cvor elem_;
   public Object pop()
   {
     if (elem_ == null)
         return null;
     else
     {
         Object temp = elem_.vrijednost_;
         elem_ = elem_.slijedeci_;
         return temp;
     }
   }
   public boolean jelPrazan()
   {
   return (elem_ == null);
   }
   public void push(Object o)
   {
     elem_ = new Cvor(o, elem_);
   }
}
```

Stack implementacija 1


```
class Test1
{
    public static double suma(MojStack a)
    {
      double suma = 0.;
      while(!a.jelPrazan())
      {
          Number br = (Number) a.pop();
          suma += br.doubleValue();
      }
      return suma;
    }
    public static void main(String[] args)
    {
      MojStack kont = new MojStack();
      kont.push(new Integer(2));
      kont.push(new Double(3.4));
      kont.push("string");
      System.out.println(suma(kont));
    }
}
```

```
$ java Test1
Exception in thread "main" java.lang.ClassCastException: java.lang.String
        at Test1.suma(Test1.java:10)
        at Test1.main(Test1.java:21)
```

Stack implementacija 1

- Ovaj pristup ima dva problema:
  - Mora se koristiti kastiranje prilikom dohvata vrijednosti
  - Nema provjere na greške – mogu se dodati vrijednosti bilo koje klase
- Prednosti generičkog kodiranja:
  - Snažnija provjera tipa tokom kompajliranja
    - Java kompajler primjenjuje strogu provjeru tipa nad generičkim kodom i izbacuje greške ukoliko kod narušava sigurnost tipova
  - Eliminiše se kastiranje
  - Omogućava implementaciju generičkih algoritama
    - Programeri mogu implementirati generičke algoritme koji rade nad kolekcijom različitih tipova, koji mogu biti prilagođeni, tipski su sigurni i jednostavniji za čitati

Java generičke klase

- Slično kao C++ template-si java od verzije 1.5 podžava koncept generičkih klasa i funkcija unutar klasa tzv *generics.*
- Da bi se kreirala generička klasa nakon imena klase unutar zagrada <> potrebno je specificirati jedan ili više generičkih tipova (npr <T,F>) koji će se koristiti u klasi.
- Dalje se taj tip (ili više njih) može koristiti bilo gdje u kodu generičke klase uz određena ograničenja.
- Tip može biti bilo koji ne-primitivni tip: klasa, interfejs, niz ili druga varijabla tipa
- Po konvenciji su tipovi jedno veliko slovo, a najčešće korišteni nazivi su:
  - E – Element
  - K – Key
  - N – Number
  - T – Type
  - V - Value

Stack implementacija 2


```
class MojStack<T>
{
    class Cvor<E>
    {
      Cvor<E> slijedeci_;
      E vrijednost_;
      public Cvor(E v, Cvor<E> s)
      {
          slijedeci_ = s;
          vrijednost_ = v;
      }
    }
    private Cvor<T> elem_;
    public T pop()
    {
      if (elem_ == null)
          return null;
      else
      {
          T temp = elem_.vrijednost_;
          elem_ = elem_.slijedeci_;
          return temp;
      }
    }
    public boolean jelPrazan()
    {
      return (elem_ == null);
    }
    public void push(T o)
    {
      elem_ = new Cvor<T>(o, elem_);
    }
}
```

Stack implementacija 2


```
class Test1
{
    public static double suma(MojStack<Number> a)
    {
      double suma = 0;
      while(!a.jelPrazan())
      {
          Number br = a.pop();
          suma += br.doubleValue();
      }
      return suma;
    }
    public static void main(String[] args)
    {
      MojStack<Number> kont = new MojStack<Number>();
      kont.push(2);
      kont.push(3.4);
      // kont.push("string"); kompajler javlja gresku
      System.out.println(suma(kont));
    }
}
```

- Od verzije 1.5 java kompajler vrši automatsku konverziju iz primitivnih tipova u njihove odgovarajuće wrapper tipove i obrnuto (autoboxing)

Instanciranje generičkih klasa

- Instanciranje objekata od generičkih klasa vrši se na slijedeći način:
  - `ImeKlase<Tip> v = new ImeKlase<Tip>(...)`
  - Kreiran objekat *v* tipa generičke klase *ImeKlase*
- Za razliku od C++ templates Java kompajler će kreirati samo jednu instancu generičke klase bez obzira na broj instanciranja klase sa različitim generičkim parametrima:

```
MojStack<Number> kont1 = new MojStack<Number>();
MojStack<String> kont2 = new MojStack<String>();
System.out.println( kont1 instanceof MojStack);\\true
System.out.println( kont2 instanceof MojStack);\\true
```

- Prilikom korištenja metoda generičkih klasa java kompjler prijavljuje sve ilegalne konverzije tipova kao greške (tj onemogućava generiranje bytecodea)

Interna pohrana objekata

- Kada god se definiše generički tip, automatski se obezbjeđuje odgovarajući sirovi tip (*raw type*)
  - naziv generičkog tipa, kod kojeg su uklonjeni parametri tipa
  - tipovi varijabli su obrisani (*type erase*) i zamijenjeni svojim graničnim tipovima (ili Object varijablama ako nema ograničenja tipa)
  - `MojStack rawStack = new MojStack();`
  - `MojStack` je sirovi tip generičkog tipa `MojStack<T>`
- Dodjeljivanje parametriziranog tipa sirovom tipu je dozvoljeno
  - `MojStack<String> mojStack = new MojStack<>();`
  - `MojStack rawStack = mojStack;  // OK`
- Dodijelom sirovog tip parametriziranom tipu, dobija se upozorenje
  - `MojStack rawStack = new MojStack();`
  - `MojStack<Integer> intStack = rawStack;`

    `// warning: unchecked conversion`
  - kompajler nema dovoljno informacija o tipu kako bi izveo sve provjere tipa koje su neophodne da se osigura sigurnost koda

Interna pohrana objekata

- Kada se poziva generički metod, kompajler ubacuje cast kada je povratni tip obrisan
  - `MojStack<Employee> buddies = . . .;`
  - `Employee buddy = buddies.pop();`
- Brisanje `pop()` ima povratni tip Object
- Kompajler prevodi poziv metoda u dvije instrukcije virtuelne mašine:
  - poziv sirovog metoda `MojStack.pop`
  - kastiranje vraćenog Object u `Employee` tip
- Kastiranje se također ubacuje kada se pristupa generičkom polju (ako bi bio public)
- Ukoliko se trebaju skupljati objekti parametriziranog tipa, treba koristiti ArrayList:
  - `ArrayList<ImeKlase<T>>`

Nasljeđivanje parametra tipa

- Ako se generički tip koristi prosljeđivanjem Number kao argumenta, svako uzastopno pozivanje `push` će biti dozvoljeno ukoliko je argument kompatibilan sa Number
  - `MojStack<Number> stack = new MojStack<Number>();`
  - `stack.add(new Integer(10));  // OK`
  - `stack.add(new Double(10.1));  // OK`
- Međutim, metod `public static double suma(MojStack<Number> a)`
  - Neće raditi sa `MojStack<Integer>` ili `MojStack<Double>`
  - `MojStack<Integer>` i `MojStack<Double>` nisu podtip od `MojStack<Number>`

Generički metodi


```
class Test2 {
  public static <K> K max(MojStack<K> kont)
  {
    K max = kont.pop();
    while(!kont.jelPrazan()) {
          K temp = kont.pop();
          if ( max.compareTo(temp) < 0 )
              max = temp;
    }
    return max;
  }
```

- Metod *max* je generički metod parametriziran tipom *K.* Uzima *kont* tipa *MojStack<K>* a vraća tip *K.*
- *Problem:*
  - Varijabla max je tipa K, što znači da može biti objekat prozvoljne klase
  - Kako znamo da klasa kojoj pripada K ima `compareTo` metod?
- *Rješenje:*
  - ograničiti tip T na klasu koja implementira `Comparable` interface
  - Postiže se postavljanjem gornjeg ograničenja za generički tip K
  - `public static <K extends Comparable> K max(MojStack<K> kont)`

Gornje ograničenje na generički tip

- Parametar tipa može imati i više granica, npr. mora biti subklasa od `Number` i da mora implementirati interface `Comparable`.
- Ukoliko je jedna od granica klasa mora biti specificirana prva
- Kompajler sam zaključi tip za *K* prilikom analize ulaznih argumenata.

```
class Test2 {
  public static <K extends Number & Comparable<K>> K max(MojStack<K> kont)
  {
    K max = kont.pop();
    while(!kont.jelPrazan()) {
          K temp = kont.pop();
          if ( max.compareTo(temp) < 0 )
              max = temp;
    }
    return max;
  }
  public static void main(String[] args) {
    MojStack<Integer> kont = new MojStack<Integer>();
    kont.push(2);
    kont.push(30);
    kont.push(4);
    int a = max(kont);
    System.out.println(a);
  }
}
```

Gornje ograničenje na generički tip


```
class Test3 {
    public static double suma(MojStack<? extends Number> a) {
      double suma = 0;
      while(!a.jelPrazan()) {
          Number br = a.pop();
          suma += br.doubleValue();
      }
      return suma;
    }
    public static void main(String[] args) {
      MojStack<Integer> kont = new MojStack<Integer>();
      kont.push(2);
      kont.push(3);
      System.out.println(suma(kont));
    }
}
```

- Prethodna verzija funkcije suma ne bi radila za poziv u Test2 main-u jer *MojStack<Integer>* nije subklasa od *MojStack<Number>*
- Uvodi se wildcard tip, gornje ograničenje tipa sa wildcard-om kako bi se relaksirala ograničenja na varijable
- *MojStack<? extends Number>* ima značenje: MojStack koji sadrži elemente koji su tipa bilo koje izvedene klase od klase Number.
- Tip MojStack<Integer> je podtip od MojStack<? extends Number>

Gornje ograničenje na generički tip

- Može li se wildcard koristiti za korupciju, npr. *MojStack<Manager>* kroz *MojStack<? extends Employee>* referencu?
  - `MojStack<Manager> menadzeri = new MojStack<Manager>();`
  - `MojStack<? extends Employee> wildcardMenadzeri = menadzeri; // OK`
  - `wildcardMenadzeri.push(lowlyEmployee); // compile-time error`
- Metodi MojStack<? extends Employee> izgledaju ovako:
  - `? extends Employee pop()`
  - `void push(? extends Employee)`
- Ovo čini nemogućim da se pozove push() metod.
- problem nemamo sa pop() sasvim je legalno dodijeliti povratnu vrijednost od pop() Employee referenci

Gornje ograničenje na generički tip

- Wildcardi se koriste za kreiranje odnosa između generičkih klasa ili interfejsa
- Zajednički roditelj od List<Number> i List<Integer> je List<?>
- Sa ciljem kreiranja odnosa između ovih klasa tako da kod može pristupiti Number-ovom metodu kroz elemente List<Integer>, koristi se gornje ograničenje sa wildcardom
  - `List<? extends Integer> intList = new ArrayList<>();`
  - `List<? extends Number>  numList = intList;`
  - `// OK. List<? extends Integer> je podtip od List<? extends Number>`

Donje ograničenje na generički tip


```
class Test4 {
    public static <K> void prenesi(MojStack<K> a, MojStack<? super K> b)
    {
       while (!a.jelPrazan())
       {
         K temp = a.pop();
         b.push(temp);
       }
    }
    public static <K> void kopiraj(K[] izvor, MojStack<K> dest)
    {
       for( int i = 0; i < izvor.length;++i)
         dest.push(izvor[i]);
    }
    public static void main(String[] args)
    {
       Double[] a = {1., 10., 2., 8.};
       MojStack<Double> kont = new MojStack<Double>();
       kopiraj(a,kont);
       MojStack<Number> dest = new MojStack<Number>();
       prenesi(kont,dest);
    }
}
```

- *MojStack<? super K>* ima značenje: MojStack koji sadrži elemente koji su tipa bilo koje bazne klase od klase *K.*

Donje ograničenje na generički tip

- Wildcard sa ograničenjem nadređenog tipa daje suprotno ponašanje od wildcarda sa ograničenjem podređenog tipa
  - Mogu se proslijediti parametri metodima, ali se ne mogu koristiti povratne vrijednosti
- MojStack<? super Manager> ima metode
  - `void push(? super Manager) i`
  - `? super Manager pop()`
- Kompajler ne zna tačan tip push metoda ali ga može pozvati sa bilo kojim Manager objektom, ali ne i sa Employee
- Međutim kada se poziva pop nema garancije o tipu vraćenog objekta. Može se dodijeliti samo Object-u
- Zaključak:
  - wildcards sa ograničenjem nadređenog tipa (donjim ograničenjem na generički tip) dopuštaju pisanje u generički objekat,
  - wildcards sa ograničenjem podređenog tipa (gornjim ograničenjem za generički tip) dopušta čitanje iz generičkog objekta.

Ograničenja na varijable generičkog tipa

- Na objektima koji su generičkog tipa mogu se sprovoditi samo one operacije koje bi bile validne za objekte tipa Object klase ili ukoliko postoje ograničenja na generički tip (bounds) one operacije koje su definirane za tip (ili više njih) korišten u specifikaciji ograničenja na generički tip.
- Isto tako nije moguće:
  - Kreirati objekte generičkog tipa (npr new T())
  - Kreirati nizove generičkog tipa
  - Koristiti generičke tipove u `static` izrazima
  - Ne može se koristiti kastiranje ili instanceof sa parametriziranim tipom
  - Ne mogu se kreirati, obuhvatiti ili izbaciti objekti parametriziranih tipova
  - Metod ne može obuhvatiti instancu parametra tipa
  - Klasa ne može imati dva overload-ana metoda koja će imati istu signaturu nakon brisanja tipa