

Objektno orijentisano programiranje 2

Generici



Uvod

- Namena kao i u C++:
 - uopštavanje apstrakcija - parametrizovani tipovi i postupci
- Često korišćenje kod zbirki (Javina biblioteka)
- Sličnost sa C++ genericima – samo namena
- Implementacija u Javi je potpuno različita:
 - nezavisno prevođenje
 - samo jedan tip bez obzira na broj poziva (konkretizacija) generika
 - veličina međukoda ne raste u zavisnosti od broja poziva
 - parametrizacija primitivnim tipom nije moguća
- Generici: tipovi (klase i interfejsi), metodi i konstruktori

Motivacija

- Do Java 1.5:
 - tip `Object` korišćen kao opšti tip
- Problem kod dohvaćanja elemenata iz heterogene zbirke
 - zbirka sadrži objekte koji pripadaju jednoj hijerarhiji tipova
 - rezultat dohvaćanja iz zbirke mora da se konvertuje naniže
 - nebezbedna konverzija
 - ako nije dohvaćen objekat očekivanog tipa – `BadCastException`
- Java 1.5 uvodi generike koji rešavaju problem
 - nebezbedna konverzija naniže više nije potrebna

Primer problema

- Pre Java 1.5:

- klasa `Stek` koristi reference tipa `Object`
 - kako bi omogućila rad sa objektima proizvoljnog tipa
- ako je u programu potreban stek diskova

```
Stek stek = new Stek();
```

```
...
```

```
Disk disk = new Disk()
```

```
String niska = "Disk";
```

```
stek.stavi(disk); stek.stavi(niska);
```

```
...
```

```
Disk disk = (Disk)stek.uzmi();
```

- problem je u nebezbednoj konverziji

Rešenje problema

- Od Java 1.5:

- generička klasa `Stek`, zavisi od tipa elemenata `T`

```
class Stek <T> {  
    public void stavi(T t) {...};  
    public T uzmi() {...};  
}
```

- stvaranje objekta steka diskova:

```
Stek<Disk> stekDiskova = new Stek<Disk>();
```

Rešenje (2)

- Stavljanje na stek diskova:

```
Disk disk1 = new Disk();
```

```
stekDiskova.stavi(disk1);
```

- prevodilac dodaje proveru tipa pri stavljanju na stek
- dozvoljeno je da bude `Disk` ili izvedena klasa iz klase `Disk`

- Uzimanje sa steka diskova:

```
Disk disk2 = stekDiskova.uzmi();
```

- nepotrebna nebezbedna konverzija naniže
- rezultat dohvatanja je sigurno tipa `Disk`

Priroda generika

- Kod generika se prevodi nezavisno
- Samo jedna klasa, bez obzira na broj poziva generika
- Konkretizovane klase: `Stek<Disk>` i `Stek<String>`, u stvari su ista klasa `Stek`
- Eksperiment koji potvrđuje navedenu činjenicu:

```
Stek<Disk> stekDiskova= new Stek<Disk>();  
Stek<String> stekNiski= new Stek<String>();  
boolean isto = (stekDiskova.getClass() ==  
                stekNiski.getClass());
```

- rezultat: `isto==true`

Provera i “brisanje” tipa

- Formalni parametar generičkog tipa, u celom telu se zamenjuje tipom `Object`
- Prevodilac ubacuje proveru tipova
 - neće dozvoliti da se na `Stek` stavi nešto što nije `Disk`
- Prevodilac ubacuje i konverziju tipa `Object` u ciljni tip
 - iako je u pitanju konverzija naniže, sada je bezbedna
- Nakon toga – brisanje informacija o tipu (*type erasure*)
- U *bytecode* – jedinstveni tip za sve pozive generika
 - dati tip se naziva “sirovim” tipom (*raw type*)

Sirovi tip

- Za klasu `class Stek <T> {...}` sirovi tip je `Stek`

- Dozvoljeno bi bilo pisati:

```
Stek<String> stekNiski = new Stek<String>();  
Stek stek = stekNiski;
```

- Razlog za sirove tipove

- kompatibilnost sa zatečenim bibliotečkim klasama (pre JDK 5.0), kao što su klase zbirki (*Collections*), koje su bile ne-generičke
- kada se koriste sirovi tipovi generičkih tipova, staro ponašanje:
 - prevodilac ne ubacuje proveru tipa pri stavljanju na stek
 - dohvatanje sa steka vraća objekat tipa `Object`

Definisanje generičke klase

- Generička klasa
 - može da bude parametrizovana jednim ili pomoću više tipova koji se pojavljuju kao formalni argumenti (parametri) generika
- ```
class ime <T1, T2, ..., Tn> {...}
```
- Parametri  $T_i$  mogu da se pojave samo u nestatičkom kontekstu:
    - u deklaraciji nestatičkog polja
    - kao povratni tip nestatičkog metoda
    - kao tip parametra nestatičkog metoda ili konstruktora
    - u deklaraciji lokalne promenljive nestatičkog metoda
    - u deklaraciji unutrašnje (nestatičke) klase ili interfejsa
  - Doseg parametra:
    - do kraja generičkog tipa, metoda ili konstruktora

# Ograničenja u definicijama

- Formalni parametar generika ne može da se koristi kao:
  - tip statičkog polja generičke klase
  - u statičkim metodima za:
    - povratni tip, tip parametra metoda, tip lokalne promenljive
  - u statičkim inicijalizacionim blokovima date klase
  - razlog: postoji samo jedna klasa (sirovi tip)
- Statičkom članu klase ne može da se pristupi preko parametrizovanog imena tipa
  - na primer, ako je metod `m()` statički metod generičke klase `A<T>`: nije dozvoljeno `A<Integer>.m()`, dozvoljeno je `A.m()`
  - slično je sa literalom klase: nije dozvoljeno `A<Integer>.class`, dozvoljeno je `A.class`
- U `A<T>` nije dozvoljeno stvoriti objekat tipa `T`, niti niz objekata tipa `T`
  - na primer, nije dozvoljeno `new T()` niti `new T[100]`
  - može: `T[] niz = (T[]) new Object[100];` (uz upozorenje)

# Korišćenje generika

- Konkretizovani tip ili “poziv” generika:
  - generički tip kojem su formalni parametri tipa zamenjeni stvarnim argumentima tipa
- Primer:
  - za generički tip `class G<T> {...}`,  
`G<String>` je poziv `G` za konkretan tip `String`
- Pozivi generika se koriste u definicijama objekata
  - primer: `G<String> gS = new G<String>();`
- Argumenti generika mogu da budu
  - klase, interfejsi i nizovi (čak i sa elementima primitivnog tipa),
- Argumenti generika ne mogu da budu
  - primitivni tipovi

# Zaključivanje o tipu

- Problem:
  - u definiciji objekta – dva puta se navode argumenti generika
  - što više parametara generika, to veći problem
- Java 7 rešava problem “dijamant simbolom”
  - u pozivu konstruktora mogu da se izostave argumenti tipa
  - primer: `G<String> gS = new G<>();`
- Prevodilac automatski zaključuje o tipovima argumenata
  - na osnovu deklaracije tipa reference sa leve strane =

# Generici i nizovi

- Argument generika može da bude niz
  - referenci na objekte
  - podataka primitivnog tipa
- Niz objekata konk. generičke klase ne može da se stvori:
  - `G<Arg> [] gNiz = new G<Arg>[n]; //!GRESKA prev.`
- Prevodi se uz upozorenje, ali baca `ClassCastException`:
  - `G<Arg> [] gNiz = (G<Arg> []) new Object[n];`
- Može (prevodi se uz upozorenje i izvršava):
  - `G<Arg> [] gNiz = (G<Arg> []) new G[n];`

# Generici, izvođenje i ugnežđenje

- Generički tip može da bude izveden iz
  - negeneričkog tipa
  - konkretizacije generičkog tipa
    - konkretizacija generičke osnovne klase može da ima kao argument neki parametar generika, na primer:  
`class GD<T1,T2> extends GB<T1> {...}`
  - sirovog tipa, na primer: `GD<T> extends GB {...}`
- Generička klasa ne može da bude potklasa `Throwable`
- Kod ugnežđenih tipova
  - i spoljašnji i ugnežđeni tip mogu da budu generički ili ne

# Ograničenje parametra

- Parametar generika može da se ograniči sa gornje strane:
  - `class G <T extends B & I1 & I2 & ...> {...}`
- Argument koji menja T mora da bude tipa ili podtipa navedenih tipova (klase B, interfejsa I1, I2, ... )
  - klasa B može da se pojavi samo ako je generik G klasa
  - u listi tipova može da bude i ranije navedeni parametar generika
- Podrazumevano ograničenje sa gornje strane je `Object`
  - `class G <T extends Object> {...}`  
je semantički isto što i `class G <T> {...}`



# Razlog za džokere

- Problem:
  - `G<IT>` nije podtip tipa `G<T>`, gde je `IT` tip izveden iz tipa `T`
  - za param. metoda tipa `G<T>`, arg. ne može da bude tipa `G<IT>`
- Različito od nizova:
  - niz `IT[]` je podtip tipa `T[]`
- Primer: ako je definisana generička klasa `Stek<T>`
  - `Stek<String>` nije jedna vrsta `Stek<Object>`
  - `String[]` jeste jedna vrsta `Object[]`
- U praksi ovo predstavlja prilično ograničenje
- Način da se ono prevaziđe – džokeri

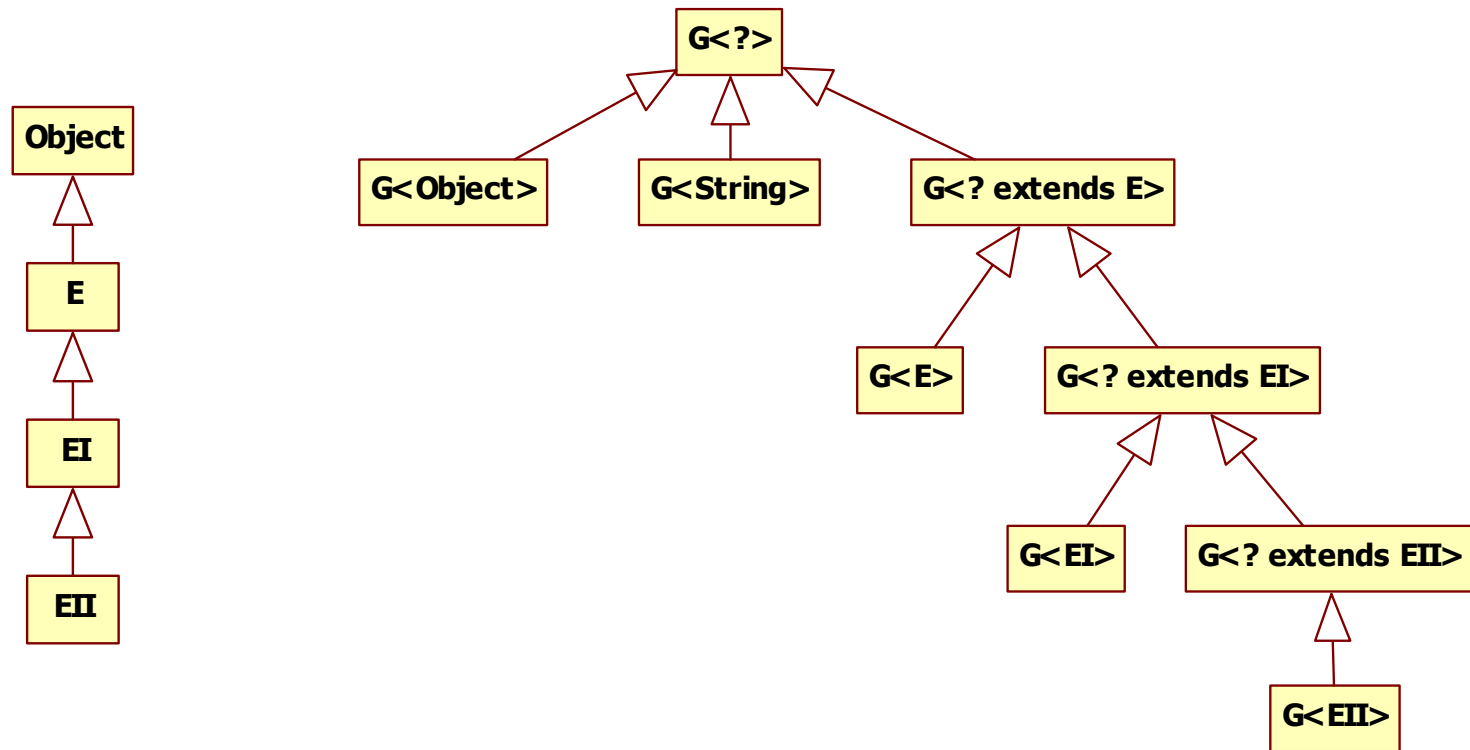
# Neograničeni džoker

- Džoker '?' na mestu stvarnog argumenta generika
  - parametrizovani tip sa kojim će da budu kompatibilni (kao podtipovi) svi parametrizovani tipovi sa prozvoljnim tipom argumenta generika
  - primer: ako je formalni parametar nekog metoda tipa `Stek<?>`, metodu mogu da se proslede stvarni argumenti tipa `Stek<String>` ili `Stek<Integer>` ili `Stek<Object>`
- Ponekad je neograničeni džoker previše opšti
  - postoje slučajevi kada se želi da se ograniči kompatibilnost parametrizovanog tipa sa njegovim podtipovima
  - postoje dve vrste ograničenja tipa: ograničenje gornje i ograničenje donje granice tipa

# Ograničenje gornje granice tipa

- Ograničenje gornje granice tipa se postiže pozivom:  
`G<? extends GGT>`
  - GGT – najviši tip u hijerarhiji tipova sa kojim postoji kompatibilnost
  - stvarni argument može da bude tipa GGT ili njegovih podtipova
- **Primer:** `Stek<? extends Disk>`
  - kompatibilan kao nadtip sa stekovima elemenata tipa `Disk` i podtipova `Disk` npr. `Stek<Disk>`, `Stek<HD>`
    - gde je `HD` izveden tip iz tipa `Disk`
  - nije kompatibilan sa stekovima elemenata drugih tipova, npr. `Stek<Object>` ili `Stek<String>` ili `Stek<Integer>`

# Hijerarhija tipova



# Ograničenje donje granice tipa

- Ograničenje donje granice tipa se postiže pozivom:  
`G<? super DGT>`
  - DGT – najniži tip u hijerarhiji tipova sa kojim postoji kompatibilnost
- **Primer:** `Stek<? super Disk>`
  - kompatibilan kao nadtip sa stekovima elemenata tipa `Disk` i njegovih nadtipova, npr. `Stek<Disk>` i `Stek<Object>`
  - nije kompatibilan sa stekovima elemenata podtipova `Disk`, niti sa stekovima drugih tipova, npr. `Stek<String>`
- **Tip** `G<? super Object>`
  - kompatibilan kao nadtip samo sa `G<Object>` pa nema svrhe

# Ograničenja ograničenih džokera

- Nije moguće da se istovremeno ograniči džoker donjom granicom i gornjom granicom tipa
  - primer: `G<? extends GGT super DGT>` nije ispravno
- Granica (gornja ili donja) ne može da se sastoji od više tipova
  - `G<? extends GGT1 & GGT2>` nije ispravno  
čak ni ako je `GGT1` neka klasa, a `GGT2` neki interfejs
  - `ni: G<? super DGT1 & DGT2>` nije ispravno

# Nizovi objekata generičke klase

- Problem: pri stvaranju niza objekata generičke klase ne može da se navede argument poziva g. klase
  - ne može:  
`G<Arg> [] gNiz = new G<Arg>[n];`
  - jedno rešenje (ranije navedeno):  
`G<Arg>[] gNiz = (G<Arg>[]) new G[n];`
  - drugo rešenje (loše):  
`G<?>[] gNiz = new G<?> [n];`
    - elementi `gNiz` mogu da budu proizvoljnog klasnog tipa
  - treće rešenje (dobro):  
`G<Arg>[] gNiz = (G<Arg>[]) new G<?>[n];`
    - preko `gNiz[i]` u niz mogu da se upišu samo objekti tipa `G<Arg>`

# Generički metodi

- Nestatički i statički metodi mogu da budu generički
- Definicija generičkog metoda:  
`<param_gen> tip_rez ime_met (param_met) {...}`
- Pozivanje konkretizacije generičkog metoda:  
`ref.<arg_gen>ime_met (arg_met) ;`  
`this.<arg_gen>ime_met (arg_met) ; // obavezno this`  
`Klasa.<arg_gen>ime_met (arg_met) ; // za statičke`
- Argumenti generičkog metoda mogu da se određuju automatski
  - poziv izgleda kao poziv negeneričkog metoda
  - arumenti generika se određuju prema tipu argumenata metoda, odnosno prema tipu konteksta u koji se vraća rezultat metoda