7

# Objektno orijentisano programiranje
## vežbe

Staša Vujičić Stanković

# Zadaci :

- Prilikom čitanja slajdova OBAVEZNO uporedo čitati zadatke sledećim redom:

1. UniverzalniDaljinski
2. Interfejs
3. kloniranje

# Interfejsi

- **Interfejs** je kolekcija konstanti i/ili apstraktnih metoda i najčešće i sadrži samo metode.
- **Interfejs definiše samo prototip metoda**.
  **U klasama koje implementiraju dati interfejs definiše se telo metoda.**
- Metode u interfejsu su uvek public i abstract, konstante uvek public, static i final
  i to se ne piše eksplicitno.
- Metode u interfejsu su po definiciji uvek instancne, tj. ne mogu biti statičke.
- Interfejs se definiše kao i klasa, jedino se umesto ključne reči class koristi ključna reč interface.

# Interfejsi

- Formalna korist interfejsa je da se definiše eksterna forma skupa metoda koji imaju određenu funkcionalnost.
- Svaki metod deklarisan unutar interfejsa **mora** da ima definiciju unutar klase koja implementira interfejs, ako želimo da kreiramo objekte te klase.
- Ukoliko se neki metod interfejsa ne definiše u klasi koja ga implementira, klasa mora biti deklarisana kao apstraktna.
- Pošto su metode u interfejsu po definiciji public, moramo koristiti ključnu reč public kada ih definišemo u odgovarajućoj klasi (atribut pristupa ne sme biti manje restriktivan).

# Interfejsi

- U Javi, klasa može da nasledi samo **jednu** klasu, ali može da implementira **više** interfejsa**.**

```
public class Klasa
        implements Klasa1, Klasa2, Klasa3
        { … }
```

- Možemo definisati interfejse bazirane jedne na drugima koristeći ključnu reč extends.

```
public interface Konverzija
        extends KonverzioniFaktori
        { … }
```

# Interfejsi

- Za razliku od klasa koje mogu da naslede samo jednu klasu, interfejs može da nasledi proizvoljan broj drugih interfejsa.

**public interface NoviInt**
**extends Int1, Int2 { … }**

// Interfejs NoviInt nasleđuje sve metode i
// konstante koje su članice interfejsa Int1 i Int2 -
// ovo je primer ***višestrukog nasleđivanja.***

# Interfejsi

- **Napomena!**
  Ako dva ili više nad-interfejsa deklarišu metod sa istim potpisom, metod mora da ima i isti povratni tip u svim interfejsima.

  U suprotnom, klasa koja implementira interfejse neće moći da implementira oba metoda,
  pošto imaju isti potpis,
  jer svaki metod u klasi mora da ima jedinstveni potpis.

# Interfejsi i polimorfizam

- Nije moguće kreirati objekte tipa interfejsa,
  ali moguće je kreirati promenljivu tipa interfejsa.

  Npr.

  **Konstante konst = null;**

  // promenljiva tipa interfejsa Konstante

# Interfejsi i polimorfizam

- Ako ne možemo kreirati objekte tipa *Konstante*, kakva korist od toga?
  Koristimo tu promenljivu za smeštanje reference na objekat proizvoljne klase koja implementira interfejs *Konstante*.
- To znači da ovu promenljivu možemo koristiti kako bismo polimorfno pozivali metode deklarisane u interfejsu *Konstante*.
- Videćemo kako to izgleda na pogodnom primeru koji se odnosi na kućnu audio/video opremu i daljinsko upravljanje.

# Pogodan primer – interfejsi i polimorfizam

- TV, hi-fi, VCR, DVD player i svaki od njih ima svoj daljinski upravljač.
- Svi daljinski upravljači verovatno imaju neki zajednički podskup dugmadi –
  power on/off, volume up, volume down, mute, itd.
- Želimo univerzalni daljinski upravljač –
  vrstu jedinstvene definicije daljinskog upravljača koji će odgovarati svoj opremi.
- Univerzalni daljinski ima mnoštvo sličnosti sa interfejsom. Sam po sebi, univerzalni daljinski ne radi ništa. On definiše skup dugmadi za standardne operacije, ali operacija za svako dugme mora biti specifično programirana da odgovara svakoj vrsti uređaja kojim želimo da upravljamo.

# Primer – univerzalni daljinski

- TV, VCR, DVD itd. možemo predstaviti klasama, a svaka od njih će koristiti isti interfejs daljinskog upravljanja – tj. skup dugmadi – ali svaka na drugačiji način.
- **RemoteControl.java**

  **public interface RemoteControl{...}**
- U interfejsu ne postoji definicija nijednog metoda.
- Metodi deklarisani u interfejsu su uvek apstraktni po definiciji, a takodje su i *public*.
- Sada, proizvoljna klasa koja zahteva korišćenje funkcionalnosti obezbeđene *RemoteControl* interfejsom samo ima da implementira *RemoteControl* interfejs i da uključi definicije za svaki od metoda iz interfejsa.

# PRIMER – UNIVERZALNI DALJINSKI

- **TV.java**

  **public class TV implements RemoteControl {...}**
- Ova klasa implementira sve metode deklarisane u interfejsu *RemoteControl*, i svaki metod ispisuje poruku tako da znamo kada je pozvan.
- **VCR.java**
- Klasa *VCR* takođe implementira *RemoteControl*.
- Naravno, možemo nastaviti i definisati klase za druge vrste uređaja koje koriste daljinski, ali ove dve su dovoljne za demonstraciju principa.

# PRIMER – UNIVERZALNI DALJINSKI

- **TestRemoteControl.java**

  **public class TestRemoteControl{...}**

- Ovo je klasa koja operiše i *TV* i *VCR* objektima preko promenljive tipa *RemoteControl*.
- Promenljiva *remote* je tipa *RemoteControl* pa je možemo koristiti za smeštanje reference na objekat proizvoljne klase koja implementira *RemoteControl* interfejs.
- Unutar *for*-petlje kreiramo na slučajan način *TV* ili *VCR* objekat.
- Pošto se tip objekta određuje u vreme izvršavanja i na slučajan način, izlaz iz programa demonstrira polimorfizam u akciji kroz promenljivu tipa interfejsa.

# Korišćenje više interfejsa

- *RemoteControl* objekat u prethodnom primeru može se koristiti za poziv samo onih metoda koji su deklarisani u interfejsu.
- Ako klasa implementira neki drugi interfejs osim *RemoteControl*, onda bi za poziv metoda deklarisanih u tom interfejsu trebalo ili
  - koristiti promenljivu tipa tog interfejsa za čuvanje reference na objekat ili
  - kastovati referencu na objekat u tip njegove stvarne klase.

# Korišćenje više interfejsa

- Neka je klasa definisana sa:

```
public class MojaKlasa implements
    RemoteControl, AbsoluteControl{
    // Definicija klase koja uključuje metode iz oba
       interfejsa
    }
```

# Korišćenje više interfejsa

- Pošto klasa implementira *RemoteControl* i *AbsoluteControl*, možemo smestiti objekat tipa *MojaKlasa* u promenljivu bilo kog od tipova ova 2 interfejsa, npr.

  **AbsoluteControl ac = new MojaKlasa();**

- Sada možemo koristiti promenljivu *ac* za pozivanje metoda deklarisanih u *AbsoluteControl* interfejsu. Međutim, ne možemo pozivati metode deklarisane u *RemoteControl* interfejsu koristeći *ac*, čak i ako referenca na objekat koju čuva ima ove metode.
- Jedna mogućnost je kastovati referencu u njen originalni klasni tip:

  **((MojaKlasa)ac).powerOnOff();**

# Korišćenje više interfejsa

- Pošto kastujemo referencu u tip *MojaKlasa*, možemo zvati proizvoljan metod definisan u toj klasi.

  Međutim, ovako ne možemo dobiti polimorfno ponašanje. Kompajler će odrediti koji metod se poziva u vreme kompajliranja.

# Korišćenje više interfejsa

- Da bismo pozivali metode iz *RemoteControl* polimorfno, moraćemo da imamo referencu sačuvanu kao taj tip (*RemoteControl*). Znajući da je objekat tipa klase koja implementira *RemoteControl* interfejs, možemo od reference smeštene u *ac* dobiti referencu tipa *RemoteControl*:

```
if(ac instanceof RemoteControl)
    ((RemoteControl)ac).mute();
```

# Korišćenje više interfejsa

- Čak i ako interfejsi *RemoteControl* i *AbsoluteControl* nisu ni u kakvoj vezi, možemo kastovati referencu iz *ac* u tip *RemoteControl*. To je moguće jer je objekat referisan pomoću *ac* zapravo tipa *MojaKlasa*, koji implementira oba interfejsa.

# Upotreba interfejsa

- Dakle, interfejs koji deklariše metode, definiše skup operacija, čime se postiže da objekti različitih klasa mogu da dele isti skup operacija.
- Ne može se kreirati objekat interfejsnog tipa, ali se može deklarisati promenljiva interfejsnog tipa koja može da čuva referencu na objekat svakog od tipova koji implementiraju taj interfejs. To znači da se promenljiva može iskoristiti za polimorfan poziv metoda deklarisanih u interfejsu.

# Kopiranje objekata (Kloniranje objekata)

- Protected metod **clone()** nasleđen iz klase Object kreira novi objekat koji predstavlja kopiju postojećeg objekta.
- Da bi kloniranje bilo moguće, neophodno je da klasa nad čijim se objektom primenjuje metod clone() implementira interfejs Cloneable.

  **public class Pas implements Cloneable { … }**

- Kada atributi originalnog objekta referišu na objekte klase, tada se samo kopiraju reference!

# Kopiranje objekata (Kloniranje objekata)

- Kada atributi tekućeg objekta referišu na objekte klasa, objekti na koje oni referišu se NE DUPLIRAJU prilikom kreiranja klona – samo se reference iskopiraju iz polja starog objekta u polja kloniranog objekta.
- To obično nije ono što želimo da se desi – i novi i stari objekat onda mogu menjati taj deljeni objekat na koji se referiše preko njihovih odgovarajućih atributa, bez registrovanja da se to dešava.

# Primer - Kloniranje objekata

- Ako dodamo klasu Buva koja nasleđuje klasu Zivotinja i definišemo je tako da implementira interfejs Cloneable, onda i bazna klasa Zivotinja mora da implementira ovaj interfejs.
- Metod clone() u klasi Buva kreira novi objekat ove klase, pri čemu se vrši kopiranje referenci *ime* i *vrsta*, tj. **ne** kreiraju se dva nova objekta klase String, već se samo kreiraju dve reference na postojeće objekte ove klase. U metodu clone() poziva se metod clone() natklase (klase Zivotinja) kako bi se iskopirala vrednost članice *vrsta*.

# Primer - Kloniranje objekata

- Dodajemo klasu PasLjubimac koja sadrži objekat klase Buva.
- Ako kloniramo objekat klase PasLjubimac, kreirana kopija sadrži referencu na isti objekat klase Buva.
  Iz tog razloga, ako se promeni ime buve iz kopije objekta klase PasLjubimac, promeniće se ime buve i u originalnom objektu ove klase.

# Komentarisanje primera

- *PasLjubimac* objekat može se kreirati sa:

  **PasLjubimac mojLjubimac =**
  **new PasLjubimac(”Maza”,”Mesanac”);**

- Ako želite istog takvog, možemo ga klonirati:

  **PasLjubimac vasLjubimac =**
  **(PasLjubimac)mojLjubimac.clone();**

- Sada imamo 2 *PasLjubimac* objekta koji sadrže referencu na isti objekat *Buva*
- Metod *clone()* će kreirati novi *PasLjubimac* objekat *vasLjubimac* i kopirati referencu na objekat *Buva* iz atributa *buva* objekta *mojLjubimac* u istoimeni atribut objekta *vasLjubimac*.

# Komentarisanje primera

- ime ljubimca može se promeniti naredbom:

  **vasLjubimac.setIme(”Lunja”);**
- Vaš pas će verovatno želeti sopstvenu buvu ☺,
  pa možemo promeniti ime njegove buve naredbom:

  **vasLjubimac.getBuva().setIme(”Buvica”);**
- Nažalost , Mazina buva će se takođe preimenovati u *Buvica*, pošto Maza i Lunja dele zajedničku buvu.

# Kopiranje objekata (Kloniranje objekata)

- Metod clone() treba implementirati na način koji nam odgovara.
- Alterantiva za metod clone() su copy-konstruktori.
- *clone()* metod klase *PasLjubimac* mogao bi biti:

```
public Object clone() throws
 CloneNotSupportedException{
  PasLjubimac ljubimac = new PasLjubimac(ime, rasa);
  ljubimac.setIme("Lunja");
  ljubimac.getBuva().setIme("Buvica");

  return ljubimac;
 }
```

- Ovde metod kreira novi PasLjubimac objekat koristeći *ime* i *rasu* tekućeg objekta.

# Kopiranje objekata (Kloniranje objekata)

- Mogli smo koristiti i nasleđeni *clone()* metod da dupliramo tekući objekat i onda da eksplicitno kloniramo član *buva* da bi referisao na nezavisan objekat

```
public Object clone() throws
CloneNotSupportedException{
    PasLjubimac ljubimac =
(PasLjubimac)super.clone();
    ljubimac.buva = (Buva)buva.clone();
    return ljubimac;
}
```

- Novi objekat kreiran nasleđenim *clone()* metodom je objekat klase *PasLjubimac*, ali se vraća kao referenca klase *Object*.
- Za pristup članu *buva* neophodna je referenca na *PasLjubimac* pa je ovde kastovanje od ključnog značaja.
- Efekat ove verzije *clone()* metoda je isti kao i prethodne verzije.