Akademija tehničko-vaspitačkih strukovnih studija
odsek NIŠ

Savremene računarske tehnologije **SRT**

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

Prof. dr Zoran Veličković, dipl. inž. el.

2019/2020.

Prof. dr Zoran Veličković, dipl. inž. el.

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

Interfejsi i polimorfizam

(9)

Sadržaj

- **POJAM INTERFEJSA**
  - Interfejsi i apstraktne klase
  - Deklaracija interfejsa
  - Podrazumevani pristup članovima interfejsa
  - Konstante u interfejsu
- **IMPLEMENTACIJA INTERFEJSA**
  - Naredba **`implements`**
  - Dodavanje metoda u implemntaciji interfejsa
  - Nasleđivanje interfejsa
  - Primena izvedenog interfejsa
- **POLIMORFIZAM I INTERFEJS U JAVI**
  - Polimorfizam klase **Pas**
- **INTERFEJSI I POLIMORFIZAM**
  - Interfejsi i polimorfizam **DU**
  - Interfejsi i polimorfizam **TV**
  - Interfejsi i polimorfizam **VCR**
  - Polimorfizam na delu

Pojam interfejsa

- Na predavanjima iz ovog predmeta već je više puta naglašeno da **METODE** definišu **NAČIN PRISTUPA** podacima u klasama.
- Upotrebom rezervisane reči **`interface`** može se potpuno **ODVOJITI NAČIN PRISTUPA PODACIMA** (kaže se interfejs) od same **REALIZACIJE KLASE**!
- Rezervisanom rečju **`interface`** se zapravo zadaje **SKUP METODA** koje će **KASNIJE** neka klasa (jedna ili više njih) **REALIZOVATI**.
- Može se **DEKLARISATI INTERFEJS**, a da se tom prilikom **NE RAZMATRA** kako će on biti zaista realizovan.
- Prilikom **REALIZACIJE INTERFEJSA**, klasa mora da napravi **POTPUN SKUP METODA** koje su njime definisane.
- **INTEFEJSOM** se dakle definiše **ŠTA** metoda treba da radi, ali **NE** i **KAKO** to treba da se izvede!
- Svaka klasa **SLOBODNO** odlučuje **KAKO** će metode iz interfejsa biti realizovane.

Interfejsi i apstraktne klase

- Imajte na umu, **DEKLERACIJA** interfejsa **NE IMPLICIRA** nikakvu realizaciju.
- Standardno, da bi metoda iz jedne klase mogla da pozove metodu iz druge, obe klase **MORAJU POSTOJATI** u trenutku prevođenja zbog **PROVERE POTPISA METODA**.
- Međutim, kreiranjem interfejsa se **PREVAZILAZI OVO OGRANIČENJE**.
- **INTERFEJSI** podržavaju **DINAMIČKO RAZREŠAVANJE METODA** u trenutku izvršavanja!
- Čak i klase koje NISU u hijerarhijskom smislu nasleđivanja **SRODNE, MOGU REALIZOVATI ISTI INTEFEJS**.
- Veoma je značajno da se **INTERFEJSI MOGU PROŠIRIVATI** baš kao i klase!
- Na osnovu već izloženog, očiglednoje da je **INTERFEJS VEOMA SLIČAN APSTRAKTNOJ KLASI**.
- Razlika je u tome što **KLASA MOŽE** da realizuje **VIŠE OD JEDNOG INTERFEJSA** !
- Sa druge strane, klasa **MOŽE DA NASLEDI SAMO JEDNU NATKLASU**.

Deklaracija interfejsa

```
Modifikator_pristupa interface ime_interfejsa
{
    tip ime_metode_1(lista_parametara_1);
    tip ime_metode_2(lista_parametara_2);
        ...
    tip ime_metode_N(lista_parametara_N);

    final tip ime_promenljive_1 = vrednost1;
    final tip ime_promenljive_2 = vrednost2;
        ...
    final tip ime_promenljive_N = vrednostN;
}
```

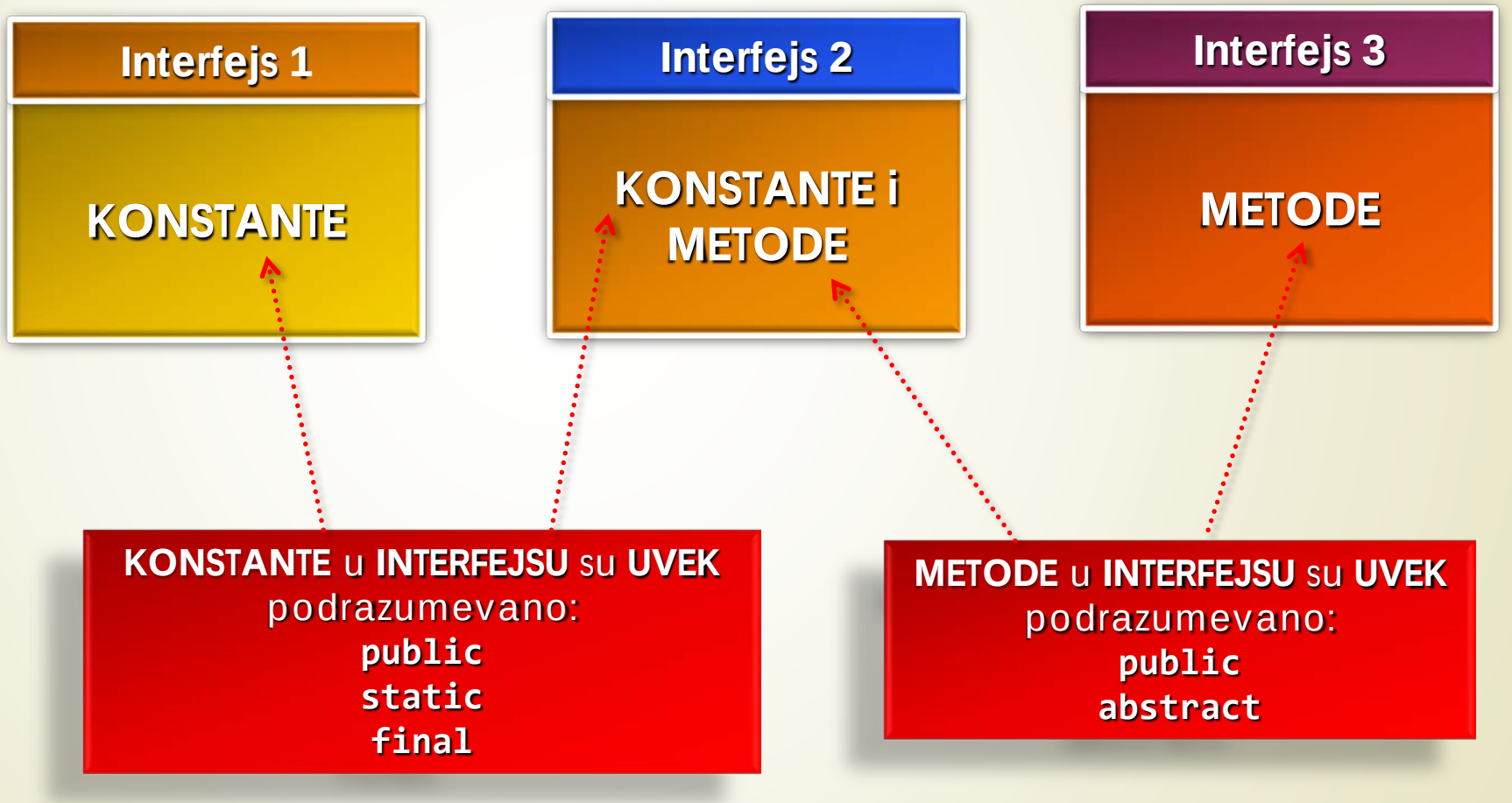
Telo interfejsa

METODE inerfejsa

PODACI intefejsa

KONSTANTE su deklarisane kao **final**

Podrazumevani pristup članovima interfejsa



Konstante u interfejsu

- Pomoću interfejsa mogu se uvesti **ZAJEDNIČKE KONSTANTE** u **VIŠE** klasa !
- Neophodno je **DEKLARISATI INTRFEJS** sa promenljivama koje su **INICIJALIZOVANE**!
- Prilikom realizacije **INTERFEJSA** sve navedene promenljive **POSTAJU KONSTANTE**:

```
interface deljeneConstante {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}
```

Implementacija interfejsa

- Kada je interfejs **DEKLARISAN**, onda, **JEDNA** Ili **VIŠE** klasa mogu da ga **REALIZUJU** (još se kaže i **IMPLEMENTIRAJU**).
- Ako klasa želi da **REALIZUJE NEKI INTERFEJS**, u njenu dekleraciju **TREBA UKLJUČITI** naredbu – ključnu reč **implements**.
- Ključna reč **implements** obaveštava kompajler o odluci - želji klase da **IMPLEMENTIRA** neki od intefejsa.

Ime klase koja **implementira** interfejs

```
modifikator_pristupa class ime_klase [extends natklasa]
[implements interfejs_1 [, interfejs_2...]] {
        // telo klase
}
```

Ključna reč

Interfejs(i) koji se implementira(ju) odvojeni zapetama

Primer interfejsa (1)

```
// Dekleracije interfejsa Callback
interface Callback
{
    void callback(int param);
}

class Client implements Callback
{
  // Implementiranje Callback interfejsa
  public void callback(int p)
    {
        System.out.println("callback pozvana sa " + p);
    }
}
```

Definicija **interfejsa Callback** koji ima samo jednu metodu **callback()**

Realizacija **interfejsa Callback** u **klasi Client**

Realizacija **metode** definisane u **interfejsu Callback**

Dodavanje metoda u implemntaciji

```
class Client implements Callback {
  public void callback(int p) {
  System.out.println("callback pozvana sa " + p);
}

  void nonInterfaceMeth() {
  System.out.println("Klase koje implementiraju" +
                     "interfaces mogu takođe " +
                     "definisati druge članove.");
  }
}
```

Metoda **callback()** **je** iz interfejsa **Callback**

Implementacija interfejsa **Callback**

Metoda **nonInterfaceMeth()** nije definisana interfejsom, ali može biti **DODATA** u procesu **Implementacije** interfejsa

Klase koje realizuju neki interfejs **MOGU DEFINISATI** i svoje **SOPSTVENE ČLANOVE**.

Nasleđivanje interfejsa

- Već je napomenuto, **INTERFEJS** se može **NASLEDITI**!
- U Javi se za **NASLEĐIVANJE INTERFEJSA** koristi rezervisana reč **`extends`** (dakle, isto kao i prilikom nasleđivanja klase).
- Kroz **NASLEĐIVANJE INTERFEJSA** mogu se **DODATI POTREBNE METODE** – slično kao i u nasleđivanju klasa.
- Ovako dobijeni **INTERFEJS** zahteva u implementaciji **REALIZACIJU SVIH METODA** – metoda iz:
  - nasleđenog interfejsa i
  - `metode` deklarisane u procesu nasleđivanja.

Primer nasleđivanje interfejsa

```
interface A {
  void metod_1();
  void metod_2();
 }
  interface B extends A {
  void metod_3();
 }
  class MojaKlasa implements B {
   public void metod_1() {
   System.out.println("Implementiran metod_1().");
  }
  public void metod_2() {
   System.out.println("Implementiran metod_2().");
  }
  public void metod_3() {
   System.out.println("Implementira metod_3().");
  }
}
```

Dekleracija **interfejsa A**

Interfejs B je **nasledio** interfejs **A** i dodao metodu **metod3()**.

Ova klasa **MORA** implementirati **sve** metode iz **A** i **B**.

Imlementacija svih metoda: **metod_1 do metod_3** iz intefejsa **A** i **B**

Primena izvedenog interfejsa

```
class Proba_Interfejsa {

   public static void main (String args[])
   {
     MojaKlasa ob = new MojaKlasa();
     ob.metod_1();
     ob.metod_2();
     ob.metod_3();
   }
}
```

Definisanje objekta klase **MojaKlasa**

Poziv metoda **metod_1()** i **metod_2()** definisanih u intefejsu **A** .

Poziv metode **meth_3()** definisane u **intefejsu B (OK)**.

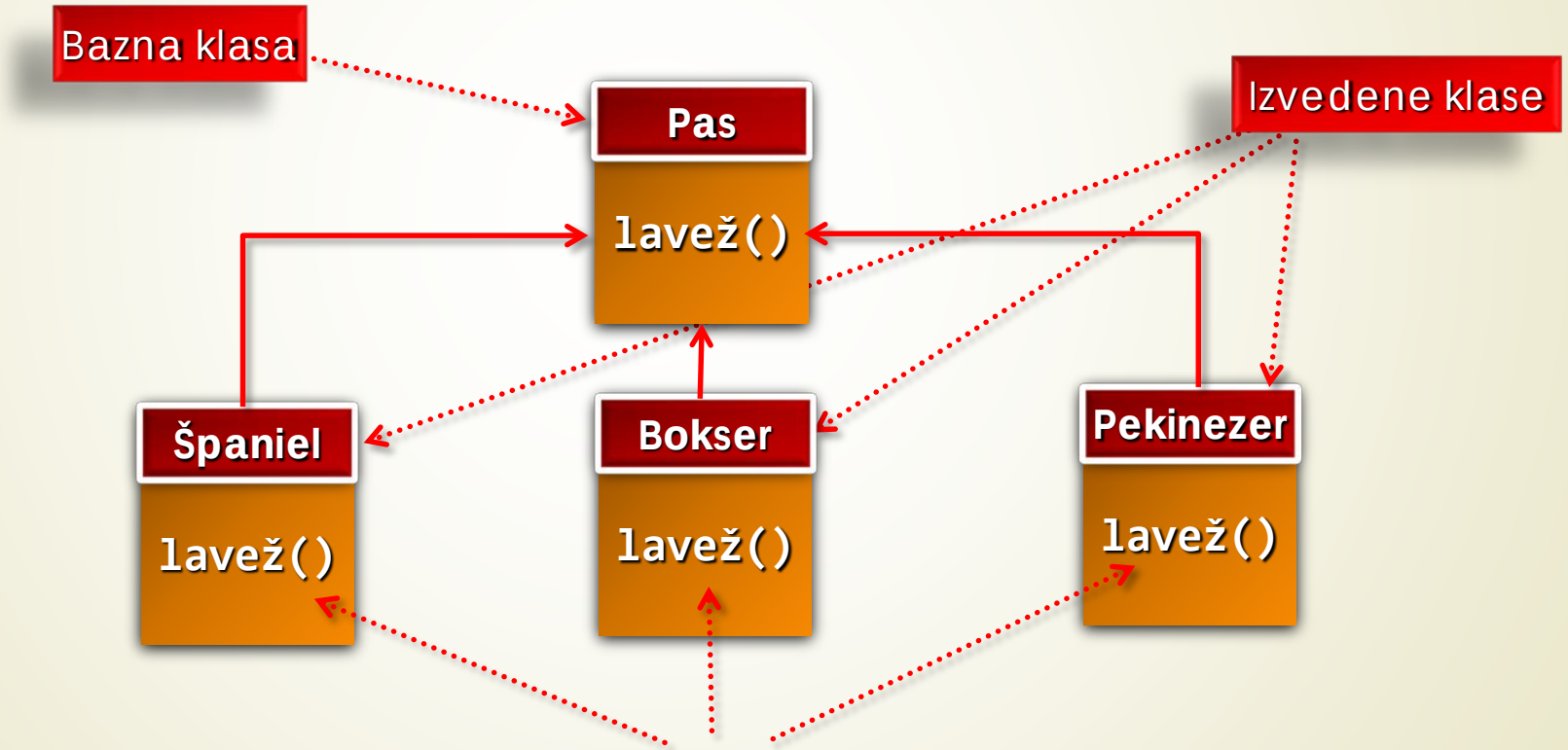
Polimorfizam u Javi

- Iz uvodnih predavanja iz ovog predmeta znamo da pojam **POLIMORFIZMA** podrazumeva mogućnost da se jedna promenljiva **BILO KOG TIPA** može koristi za referenciranje objekata **RAZLIČITIH TIPOVA**.
- Takođe, **POLIMORFIZAM** podrazumeva i **POZIVANJE ONOG METODA** koji je karakterističan za **TIP OBJEKTA** koja ta promenljiva referencira.
- Zahvaljujući polimorfizmu, **ISTI POZIV METODA** može da se ponaša **DRUGAČIJE**, u zavisnosti od **TIPA OBJEKTA** na koji se primenjuje.
- Već znamo, **POLIMORFIZAM** funkcioniše **SAMO** sa objektima **IZVEDENE KLASE**.
- **REFERENCA NA OBJEKT IZVEDENE KLASE** može da se čuva u promenljivoj tipa **IZVEDENE KLASE**, ali i u promenljivoj tipa bilo koje direktne ili indirektne bazne klase.
- **POLIMORFIZAM** se isključivo primenjuje na **METEODE**, a nikako na **PODATKE** članove.
- Da li možete obasnit zašto je ovo pravilo logično!

Polimorfizam klase Pas

```
Pas aDog;
```

// promeljiva **aDog** u kojoj se može
// čuvati **BILO KOJA** vrsta objekta **Pas**




```
aDog.lavež();
```

// poziv **BILO KOG** od ovih metoda
// zavisi od **TIPA OBJEKTA**

Interfejsi i polimorfizam (1)

- Kod kuće verovatno imate **TELEVIZOR** (TV), **DVD** plejer (VCR), **HiFi** ili slične uređaje koji poseduju **DALJINSKI UPRAVLJAČ** (DU).
- Na svim daljinskim upravljačima verovatno postoji **ZAJEDNIČKI PODSKUP DUGMADI** – **KOMANDE** koje imaju **ISTE** (ili slične) funkcije.
- **PRIMER** ovih funkcija mogu biti:
  - uključivanje,
  - isključivanje,
  - gašenje tona – mute funkcija,
  - pojačavanje i utišavanje tona,
  - promena kanala naviše,
  - promena kanala naliže,
  - ostale slične funkcije.

Interfejsi i polimorfizam (2)

- Da li se može napraviti **UNIVERZALNI DALJINSKI UPRAVLJAČ** (DU) koji se može **ADAPTIRATI** uređaju koji ga koristi?
- Sam za sebe **DU** ne služi ničemu, jer se njime **DEFINIŠE SKUP STANDARDNIH FUNKCIJA**, ali se funkcija svakog dugmeta mora programirati za svaki uređaj **PONAOSOB**.
- **SKUP UREĐAJA** se može predstaviti **KLASAMA**.
- **SVAKA KLASA** koristi **ISTI INTERFEJS - DU** ali na sebi svojstven - **DRUGI NAČIN**.

TV DU

Car Hi-Fi DU

Home Theather DU

Interfejsi i polimorfizam (3)

```
public interface RemoteControl
 {
    boolean powerOnOff();                  // Vraća novo stanje, on = true
    int volumeUp(int increment);           // Vraća novi nivo glasnoće
    int volumeDown(int decrement);         // Vraća novi nivo glasnoće
    void mute();                           // Obustavi zvučni izlaz
    int setChannel(int channel);           // Biranje i vraćanje br. kanala
    int channelUp();                       //  Vraćanje novog kanala
    int channelDown();                     // Vraćanje novog kanala
 }
```

Definisanje interfejsa **RemoteControl**

Definisanje METODA interfejsa **RemoteControl**

Interfejsi i polimorfizam TV (1)

```
import static java.lang.Math.max;
import static java.lang.Math.min;
public class TV implements RemoteControl {
    private String make = null;
    private int screensize = 0;
    private boolean power = false;
    private int MIN_VOLUME = 0;
    private int MAX_VOLUME = 100;

    public TV(String make, int screensize)
    {
        this.make = make;
        this.screensize = screensize;
}
```

Klasa **TV** Implementira **interfejsa RemoteControl**

Inicijalizovane promenljive (konstante) definisane u klasi **TV**

Konstrktor klase **TV** sa parametrima **IME UREĐAJA** i **veličine ekrana**

Interfejsi i polimorfizam TV (1)

```
public boolean powerOnOff() {
    power = ! power;
    System.out.println(make + " "+ screensize + " inch TV power
                                    " + (power ? "on. ":"off. "));
    return power;
}
```

Realizacija metode **powerOnOff()**

Umesto **prave funkcionalnosti** ispisuje se zadatak na konzoli

```
public void mute() {
    if( ! power) {          // Ako je isključen
    return;                 // ne radi ništa
   }
    volume = MIN_VOLUME;
      System.out.println(make + " "+ screensize + " inch TV volume level: "
                                                  + volume);
}
```

Realizacija metode **mute()**

Interfejsi i polimorfizam TV (2)

```
public int volumeDown(int decrement) {
    if ( ! power) {              // Ako je uređaj isključen
      return 0;                  // ne radi ništa
    }                            // U suprotnom:
    volume -= decrement;
    volume = max(volume, MIN_VOLUME);
    System.out.println(make + " " + screensize + " inch TV volume level: "
                                                        + volume);
    return volume;
  }
  . . .
}
```

Realizacija metoda **volumeDown()**

Ne manja glasnoća od najmanje definisane

Interfejsi i polimorfizam VCR (1)

```
import static java.lang.Math.max;
import static java.lang.Math.min;

public class VCR implements RemoteControl {
 public VCR(String make) {
 this.make = make;
 }

public boolean powerOnOff() {
 power =! power;
 System.out.println(make + " VCR power " + (power ? " on. " : " off."));
 return power;
}
```

Klasa **VCR** implementira interfejsa **RemoteControl**

Konstruktor klase **VCR**

Realizacija metode **powerOnOff()** u klasi **VCR**

Interfejsi i polimorfizam VCR (2)

```
public int volumeUp(int increment) {
   if(!power) {
 return 0;
 }
  volume += increment;
  volume = min(volume, MAX_VOLUME);
  System.out.println(make + " VCR volume level: " + volume);
  return volume;
 }
 . . .
}
```

Realizacija metode **volumeUP()** u klasi **VCR**

Iste metode su implementirane za **VCR** uređaj

Na sličan način se mogu kreirati klase koje realozuju **Hi-Fi** ili neki drugi uređaj!

Polimorfizam na delu

```
import static java.lang.Math.random;

public class TryRemoteControl {
public static void main(String args[]) {
RemoteControl remote = null;
for(int i = 0 ; i<5 ; i++) {
  if(random() < 0.5)
    remote = new TV(random() < 0.5 ? "Sony" : "Hitachi", random() < 0.5 ? 32 : 28);
  else
    remote = new VCR(random() < 0.5 ? "Panasonic" : "JVC");
  remote.powerOnOff();          // Prekidač je uključen
  remote.channelUp();           // Postavi sledeći kanal
  remote.volumeUp(10);          // Pojačaj glasnost na 10
  }
 }
}
```

Parametri konstruktora

Metoda iz **random** paketa

Polimorfizam na delu – moguć izlaz

```
Sony 28 inch TV power on.
Sony 28 inch TV tuned to channel: 1
Sony 28 inch TV volume level: 10
Panasonic VCR power on.
Panasonic VCR tuned to channel: 1
Panasonic VCR volume level: 10
Sony 32 inch TV power on.
Sony 32 inch TV tuned to channel: 1
Sony 32 inch TV volume level: 10
JVC VCR power on.
JVC VCR tuned to channel: 1
JVC VCR volume level: 10
Sony 28 inch TV power on.
Sony 28 inch TV tuned to channel: 1
Sony 28 inch TV volume level: 10
```

Generički nterfejs Comparable<T> (1)

- **GENERIČKI** (engl. *generics*) **TIPOVI** omogućavaju pravljenje **KLASA**, **INTERFEJSA** i **METODA** koje **BEZBEDNO RADE** sa podacima **RAZLIČITIH TIPOVA**!
- Kada se razviju **ALGORITMI** za rad sa **GENERIČKIM TIPOVIMA**, oni se mogu primeniti na **RAZLIČITE TIPOVE PODATAKA**!
- Na sledećim slajdovima prikazan je sadržaj helpa koji nudi Oracle vezano za generički interfejs **`Comparable<T>`**.
- Interfejs **`Comparable<T>`** poseduje samo jednu metodu **`CompareTo()`.**

Oracle help: interfejs Comparable<T> (2)

O **GENERIČKIM TIPOVIMA** u Javi više do kraja kursa

Oracle help: interfejs Comparable<T> (3)

```
public interface Comparable<T>
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). (interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

O **KOLEKCIJAMA** u Javi više do kraja kursa

The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.

For example, if one adds two keys `a` and `b` such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sorted set that does not use an explicit comparator, the second `add` operation returns false (and the size of the sorted set does not increase) because `a` and `b` are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with equals. One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:

```
{(x, y) such that x.compareTo(y) <= 0}.
```

The *quotient* for this total order is:

```
{(x, y) such that x.compareTo(y) == 0}.
```

It follows immediately from the contract for `compareTo` that the quotient is an *equivalence relation* on `C`, and that the natural ordering is a *total order* on `C`. When we say that a class's natural ordering is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's `equals(Object)` method:

```
{(x, y) such that x.equals(y)}.
```

This interface is a member of the Java Collections Framework.

Metoda interfejsa Comparable<T>

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
int	`compareTo(T o)` Compares this object with the specified object for order.

Jedina metoda interfejsa **CompareTo()**

Method Detail

compareTo

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all x and y. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o - the object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

Throws:

`NullPointerException - if the specified object is null`