

Siniša Nikolić

Sadržaj

- ☞ Ključna reč static,
- ☞ Upoznavanje sa mehanizmima nasleđivanja
 - ☉ Princip nasleđivanja
 - ☉ Apstraktne klase
 - ☉ Polimorfizam
- ☞ Implementacija interfejsa u Javi,
- ☞ Statičko povezivanje (*static/early binding*) i dinamičko (*dynamic/late binding*) povezivanje

Ključna reč static

☞ Definiše statičke attribute i metode

```
class StaticTest {  
    int a = 1;  
    static int i = 47;  
    static void metoda() { i++; }  
}
```

☞ **vezuju se za klasu**, a ne za objekat klase

☞ Vrednost atributa se na čuva u objektima, već se skladišti u *Field Data* prostoru koji se nalazi u *Class Data* (prostor namenjen za skladištenje metapodataka i informacija za klasu) koji pripada prostoru *Method Area* koji pripada delu memorije *Metaspace*.

☞ Statički atributi i metode postoje i bez kreiranja objekta zato im se **treba** pristupiti preko imena klase

```
StaticTest.i++;
```

☞ Statički atributi imaju istu vrednost u svim objektima

☞ Ako promenim statički atribut u jednom objektu, on će se promeniti i kod svih ostalih objekata

Ključna reč static

☞ Namena statičkih metoda:

- 🟡 pristup i rad sa statičkim atributima
- 🟡 opšte metode za koje nije potrebno da se kreira objekat

☞ Primeri upotrebe:

```
// out je staticki atribut
```

```
System.out;
```

```
Math.PI;
```

```
// ovo ostalo su staticke metode
```

```
Math.random();
```

```
Math.sin();
```

```
public static void main(String[] args) {...}
```

☞ Kada pozovemo klasu Hello

```
java Hello
```

☞ Poziva se njena funkcija main, ovo se u stvari dešava kada pokrenemo program

```
Hello.main(args)
```

Statički blok

- ☕ Statički blok se izvršava samo jednom, prilikom prvog korišćenja klase
- ☕ Unutar statičkog bloka može se pristupati samo statičkim atributima i mogu se pozivati samo statičke metode

```
class Test {  
    static int a;  
    static int b;  
        int c;  
    static void f() {  
        b = 6;  
    }  
    static {  
        a = 5;  
        // c = 1; //zabranjeno  
        f();  
    }  
}
```

Primer00

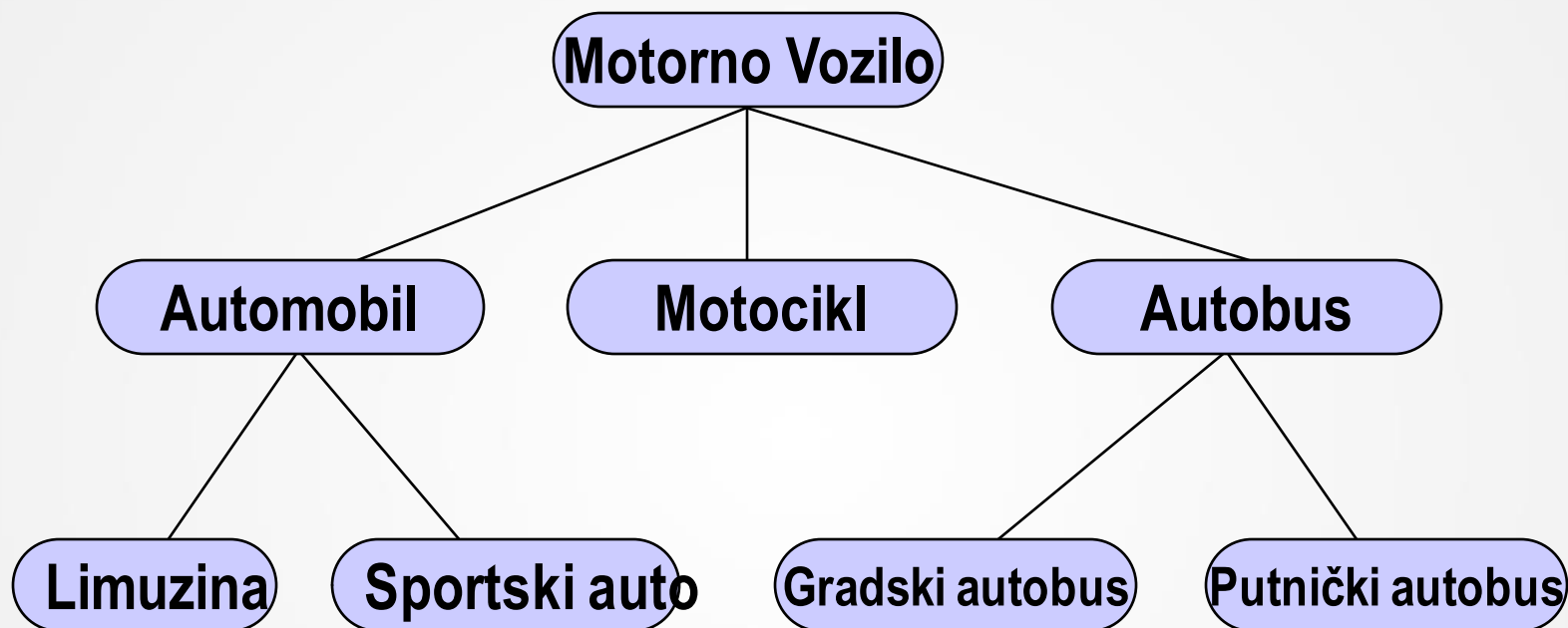
Princip nasleđivanja

- ☹ Vrlo često novi programi nastaju proširivanjem prethodnih. Najbolji način za stvaranje novog softvera je imitacija, doterivanje i proširivanje postojećeg.
- ☹ Zamislite slučaj kada imate izvorni kod neke klase. Postavite sebi pitanje, „Kako bi vi mogli da taj kod ponovo iskoristite?“ Mogli biste da ga iskopirate i u kopiji menjate ono što je potrebno.
- ☹ Kod 1 klase kopiramo na 10 mesta
- ☹ Postavite sebi pitanje "Koliki bi problem nastao ukoliko kod originalne klase sadrži neku grešku? Da li moramo tu grešku ispraviti u svim novim klasama?" Bez pažljivog planiranja završili biste sa reorganizovanom gomilom koda, prepunom bagova.

Princip nasleđivanja

- ☪ Relacija nasleđivanja omogućuje proširenje ponašanja postojeće klase.
- ☪ Generalizacija – Entiteti sa zajedničkim osobinama se grupišu tako da se njihove zajedničke osobine definišu samo jednom u osnovnoj klasi koja predstavlja njihovu generalizaciju.
- ☪ Specijalizacija – Sve ostale osobine entiteta koji su karakteristične za svaki posmatrani entitet se definišu u zasebnim klasama koje nasleđuju osnovnu klasu, te nove klase predstavljaju specijalizaciju entiteta osnovnih klasa.
- ☪ Nasleđivanje se može tumačiti kao “je vrsta” veza

Princip nasleđivanja – hijerarhija



Princip nasleđivanja

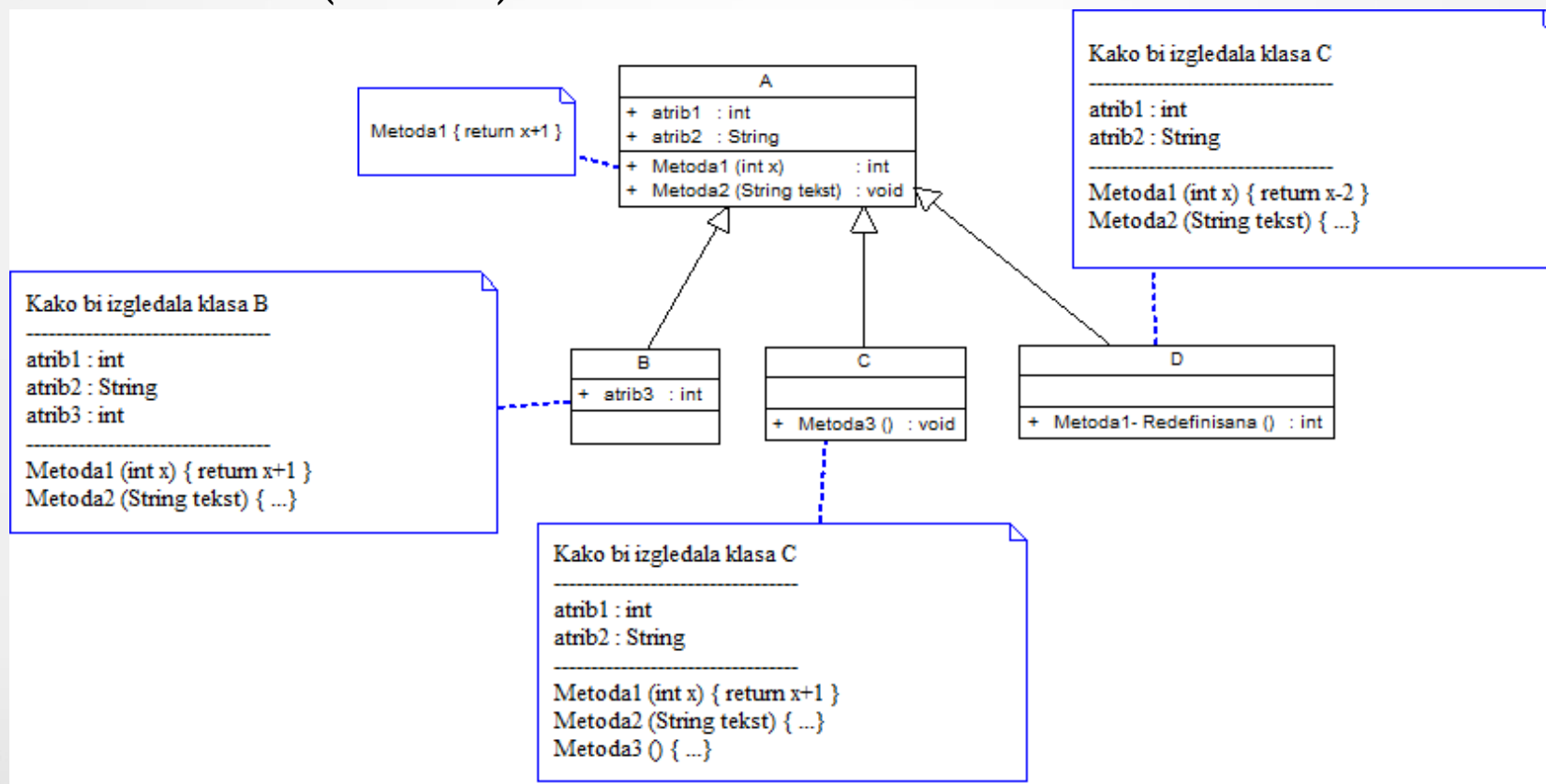
- ☪ Klasa koja nasleđuje drugu klasu (izvedena klasa) preuzima sve atribute i metode klase koju nasleđuje (osnovna klasa), efekat je sličan kao kad bi mi ručno prekopirali kod osnovne klase u izvedenoj klasi – ali nismo.
- ☪ Npr. klase B,C,D nasleđuju klasu A. Izvedene klase B,C,D (potomak, podklasa – *subclass*, dete klasa – *child class*,) predstavljaju jednu specijalnu vrstu osnovne klase A (predak, nadklasa – *superclass*, roditaljska klasa – *parent class*), gde klase B,C,D nasleđuje sve atribute i sve metode od klase A.

Princip nasleđivanja



Nova izvedena B, C ili D klasa može da:

- proširi strukturu podataka osnovne klase A dodavanjem novih atributa (klasa B)
- proširi funkcionalnost osnovne klase A dodavanjem novih metoda (klasa C)
- izmeni funkcionalnost osnovne klase A redefinisanjem postojećih metoda (klasa D)

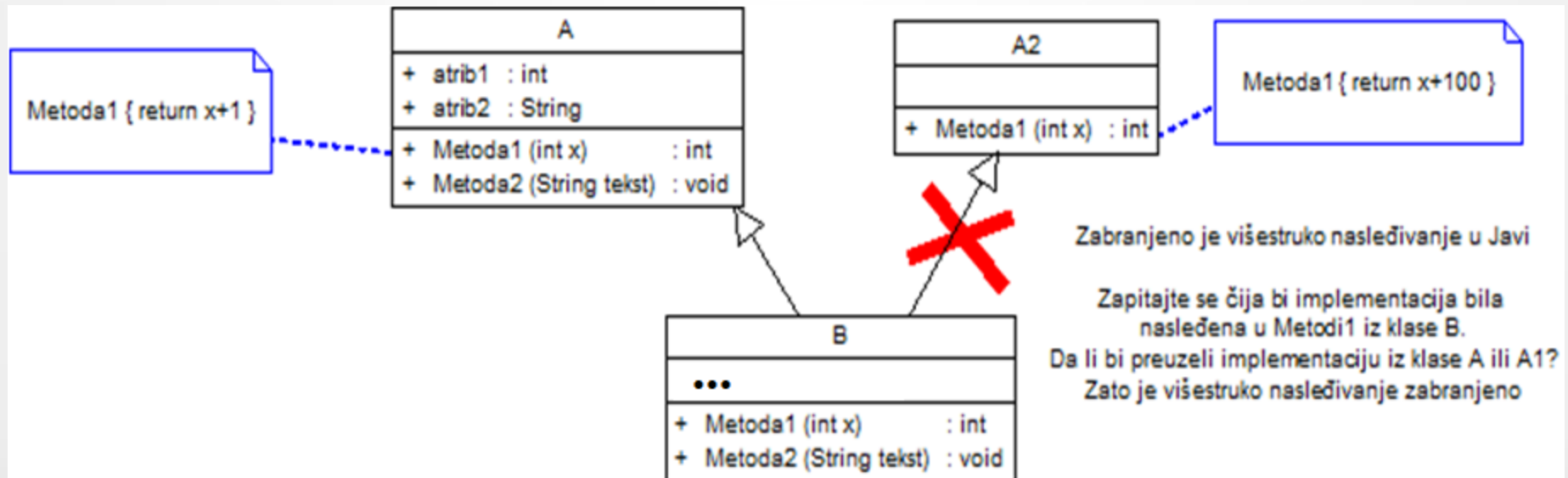


Princip nasleđivanja

- ☪ Primer nasleđivanja za osobu na fakultetu:
 - 👤 student je osoba koja studira,
 - 👤 profesor je osoba koja predaje na fakultetu.
- ☪ Imamo osnovnu klasu **osoba** (JMBG, ime i prezime, grad) i specijalizacije bi bile **student** (osoba koja ima indeks i ocene) i **profesor** (osoba koja radi na fakultetu i ima zvanje, platu, radno mesto, predmete koje drži).

Princip nasleđivanja

- ☞ Postoji samo jednostruko nasleđivanje
- ☞ Jedna klasa može samo jednu naslediti, ali više klasa može nasleđivati istu klasu
- ☞ Ako ništa ne napišemo klasa nasleđuje Object klasu
- ☞ Ključna reč **extends**



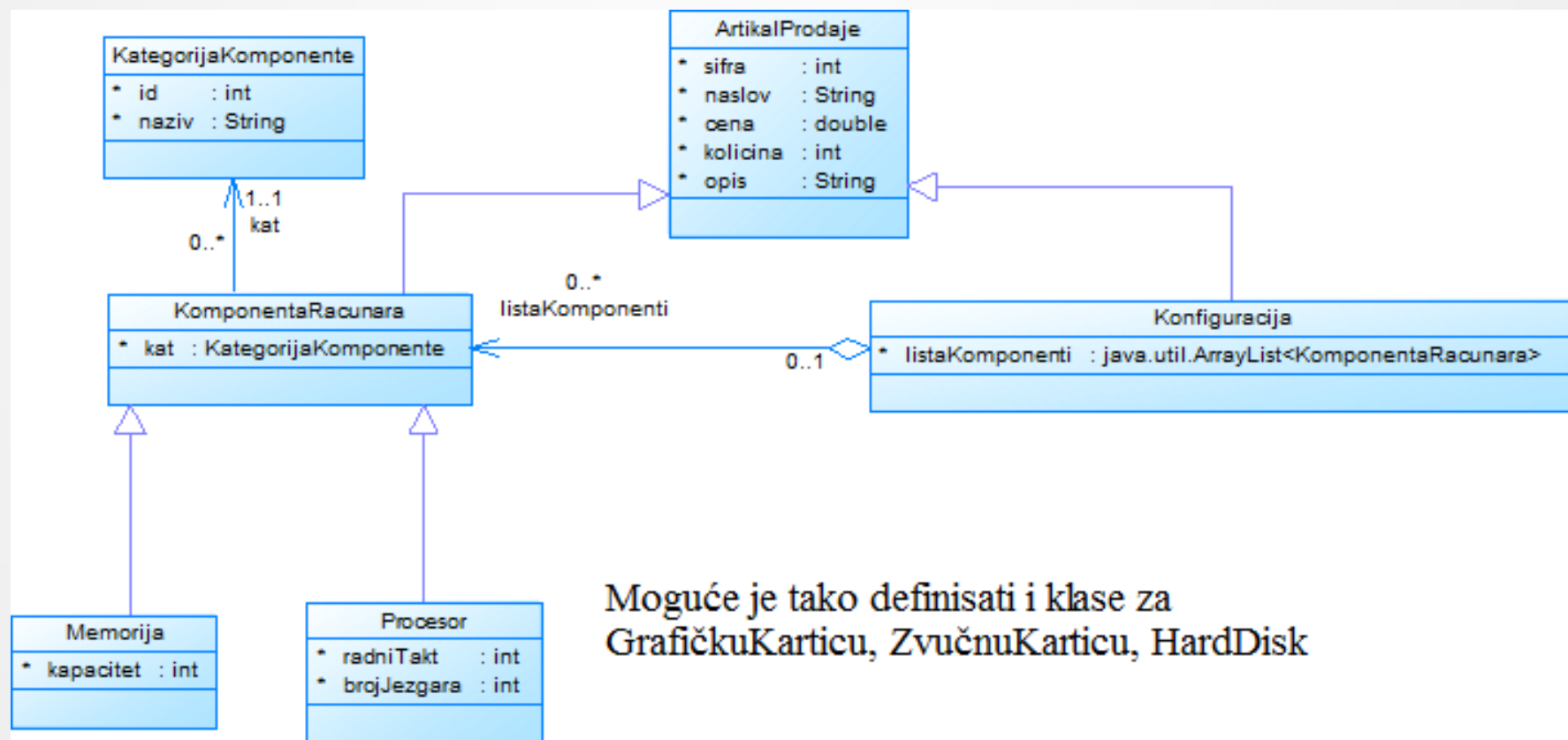
Princip nasleđivanja

☞ Primer nasleđivanja za artikle prodaje u prodavnici računara:

- Artikal Prodaje koji predstavlja generalizaciju za sve proizvode koji se mogu prodavati u prodavnici računara. Opisan je šifrom, nazivom (naslovom), cenom, raspoloživom količinom i opisom.
- Komponenta računara je artikal koji se prodaje i dodatno je opisana *kategorijom komponente*.
- Gotova konfiguracija je artikal koji se prodaje i dodatno je opisana *listom komponenti računara* koji ulaze u konfiguraciju
- Memorija računara je jedna od specijalizacija komponenti računara i dodatno je opisana *kapacitetom* memorije
- Procesor računara je jedna od specijalizacija komponenti računara i dodatno je opisan *radnim taktom* i *brojem jezgara*

Princip nasleđivanja

- ☕ Primer nasleđivanja za artikle prodaje u prodavnici računara – Dijagram klasa:



Moguće je tako definisati i klase za
GrafičkuKarticu, ZvučnuKarticu, HardDisk

Princip nasleđivanja

- ☹ Ako napišemo ključna reč **final** ispred naziva klase nasleđivanje je zabranjeno
- ☹ Nasleđivanje je zavisno od modifikatora pristupa definisanih za metode i attribute klase pretka. Oni su:
 - 🟡 vidljivi unutar metoda klasa naslednica i mogu se pozivati nad objektima klasa naslednica – public, protected
 - 🟡 nisu vidljivi unutar metoda klasa naslednica i ne mogu se pozivati nad objektima klasa naslednica – private, unspecified

Princip nasleđivanja – redefinisanje metoda

- ☪ **Method overriding** – Redefinisanje metoda je pojava da u klasi naslednici postoji metoda istog imena i parametara kao i u baznoj klasi
- ☪ Cilj je definisati/izmeniti/proširiti funkcionalnost metode roditeljske klase
- ☪ Redefinisane metode mogu se anotirati u kodu
Anotacija **@Override**
- ☪ **Primer:**
 - 🟡 klasa A ima metode **metoda1()** i **metoda2()**
 - 🟡 klasa B nasleđuje klasu A i takođe ima metode **metoda1()** i **metoda2()**, ali samo **metoda1()** je **redefinisana**

Princip nasleđivanja – redefinisanje metoda

```
class A {  
    int metoda1() {  
        System.out.println("metoda1 klase A");  
    }  
    int metoda2() {  
        System.out.println("metoda2 klase A");  
    }  
}  
  
class B extends A {  
    @Override  
    int metoda1() {  
        System.out.println("metoda1 klase B");  
    }  
}  
  
...  
A varA = new A();  
B varB = new B();  
varA.metoda1();  
varB.metoda1();  
varA.metoda2();  
varB.metoda2();
```

Princip nasleđivanja – redefinisavanje metoda

☕ Konzola:

```
metoda1 klase A
```

```
metoda1 klase B
```

```
metoda2 klase A
```

```
metoda2 klase A
```

Princip nasleđivanja – reč super

- ☞ Ključna reč **super** označava roditeljsku klasu. Ona se može koristiti i u metodama i u konstruktorima.
- ☞ Ključna reč **super** u konstruktoru označava da pozivamo konstruktor roditeljske klase. Prva linija u konstruktoru klase naslednice **mora** biti poziv konstruktora roditeljske klase
- ☞ Korišćenjem reči **super** možemo pristupiti metodama roditeljske klase koje su redefinisane

primer 01 – sa super

Apstraktne klase

- ☼ Osnovna klasa koja nema nijedan konkretan (realan) objekat, već samo predstavlja generalizaciju izvedenih klasa, naziva se apstraktnom klasom.
- ☼ Apstraktna klasa može da sadrži apstraktne funkcije, koje su u ovoj klasi samo deklarisanе, a nije implementirane
- ☼ Klase koje ne mogu imati svoje objekte, već samo njene klase naslednice mogu da imaju objekte (ako i one nisu apstraktne)

Apstraktne klase

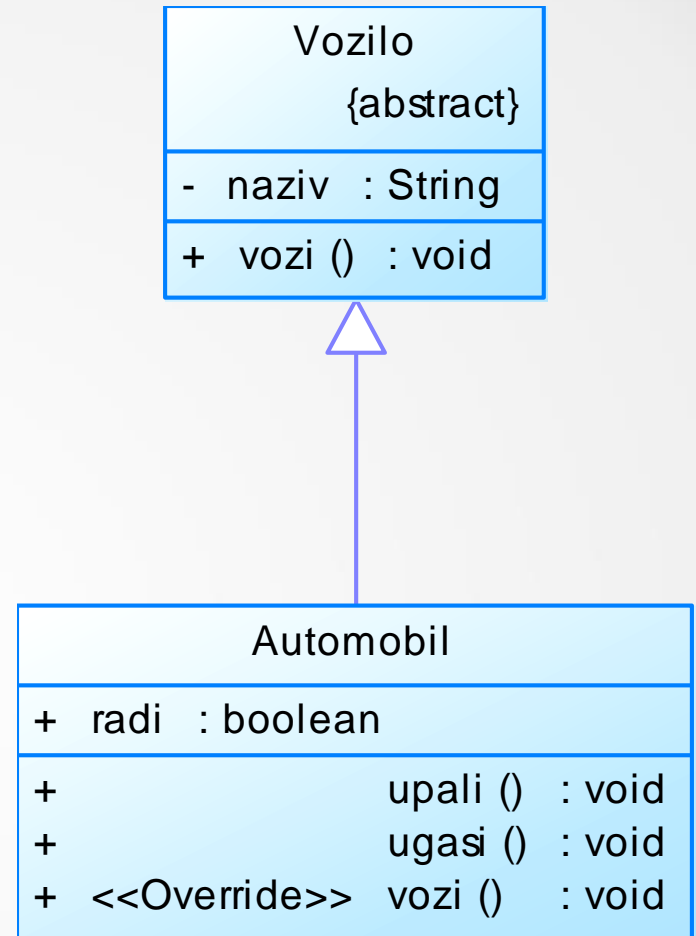
- ☞ Ako klasa ima makar jednu apstraktnu metodu, mora da se deklariše kao apstraktna.
- ☞ Apstraktna klasa ne mora da ima apstraktne metode!

```
abstract class A {  
    int i;  
    public void metoda1() { ... }  
    public abstract void metoda2();  
    ...  
}  
  
class B extends A {  
    @Override  
    public void metoda2() { ... }  
}
```

Apstraktne klase

```
public abstract class Vozilo {  
    private String naziv;  
    public abstract void vozi();  
}
```

```
public class Automobil extends Vozilo {  
    public boolean radi;  
    public void upali() {  
        radi = true;  
    }  
    public void ugasi() {  
        radi = false;  
    }  
    @Override  
    public void vozi() {  
        ...  
    }  
}
```



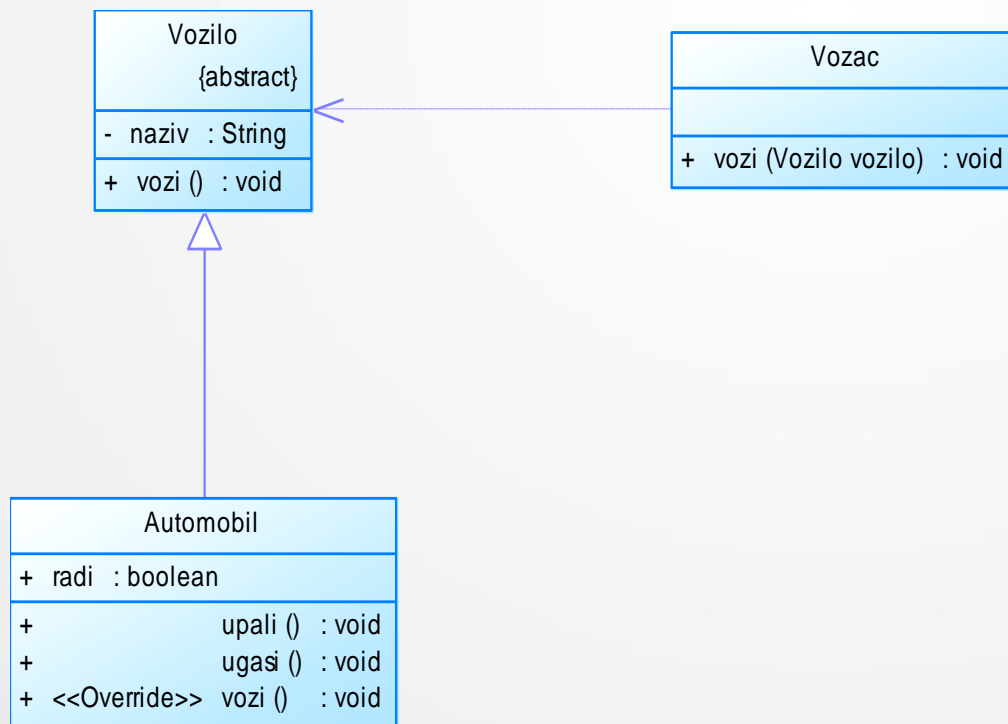
primer 02

Polimorfizam

- ☪ Opisuje koncept u kome se određena akcija može izvršiti na više načina. Polymorphism je nastao kombinacijom grčkih reči *poly* (više) i *morphs* (izgled/forma)
- ☪ Može biti:
 - 🟡 *compile time polymorphism* (implementira sa *method overloading*). Za vreme kompajliranja zna se koji se metod poziva na osnovu argumenata
 - 🟡 *runtime polymorphism* (implementira sa *method overriding*). Kompajler ne određuje koja konkretna implementacija metoda će biti pozvana. Implementacija se određuje za vreme izvršavanja u zavisnosti od toga koji je konkretan objekat instanciran.
- ☪ Naglasak na *Runtime polymorphism* koji se još zove *Dynamic Method Dispatch*.

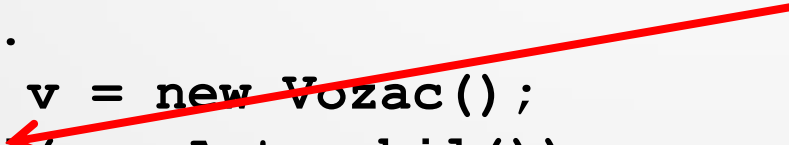
Polimorfizam

- ☕ Situacija kada se poziva metoda nekog objekta, a ne zna se unapred kakav je to konkretan objekat
 - 🟡 ono što se zna je koja mu je bazna klasa
- ☕ Tada je moguće u programu pozivati metode bazne klase, a da se zapravo pozivaju metode konkretne klase koja nasleđuje baznu klasu



Polimorfizam

```
abstract class Vozilo {  
    abstract void vozi();  
}  
  
class Automobil extends Vozilo {  
    @Override  
    void vozi() { ... }  
}  
  
class Kamion extends Vozilo {  
    @Override  
    void vozi() { ... }  
}  
  
class Vozac {  
    void vozi(Vozilo v) {  
        v.vozi();  
    }  
}  
  
...  
Vozac v = new Vozac();  
v.vozi(new Automobil());
```



Polimorfizam

- ☞ Prednost korišćenja polimorfizma ogleda se u tome da nam on omogućava kreiranje uniformnog pristupa/kontrole ka različitim objektima koji imaju zajednički podskup operacija.
- ☞ Rezultat polimorfizma je kod koji je više koncizan i lakši za održavanje.

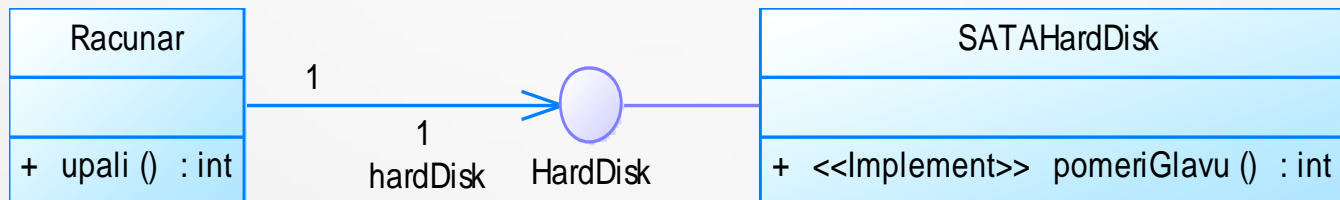
primer 03

Interfejsi

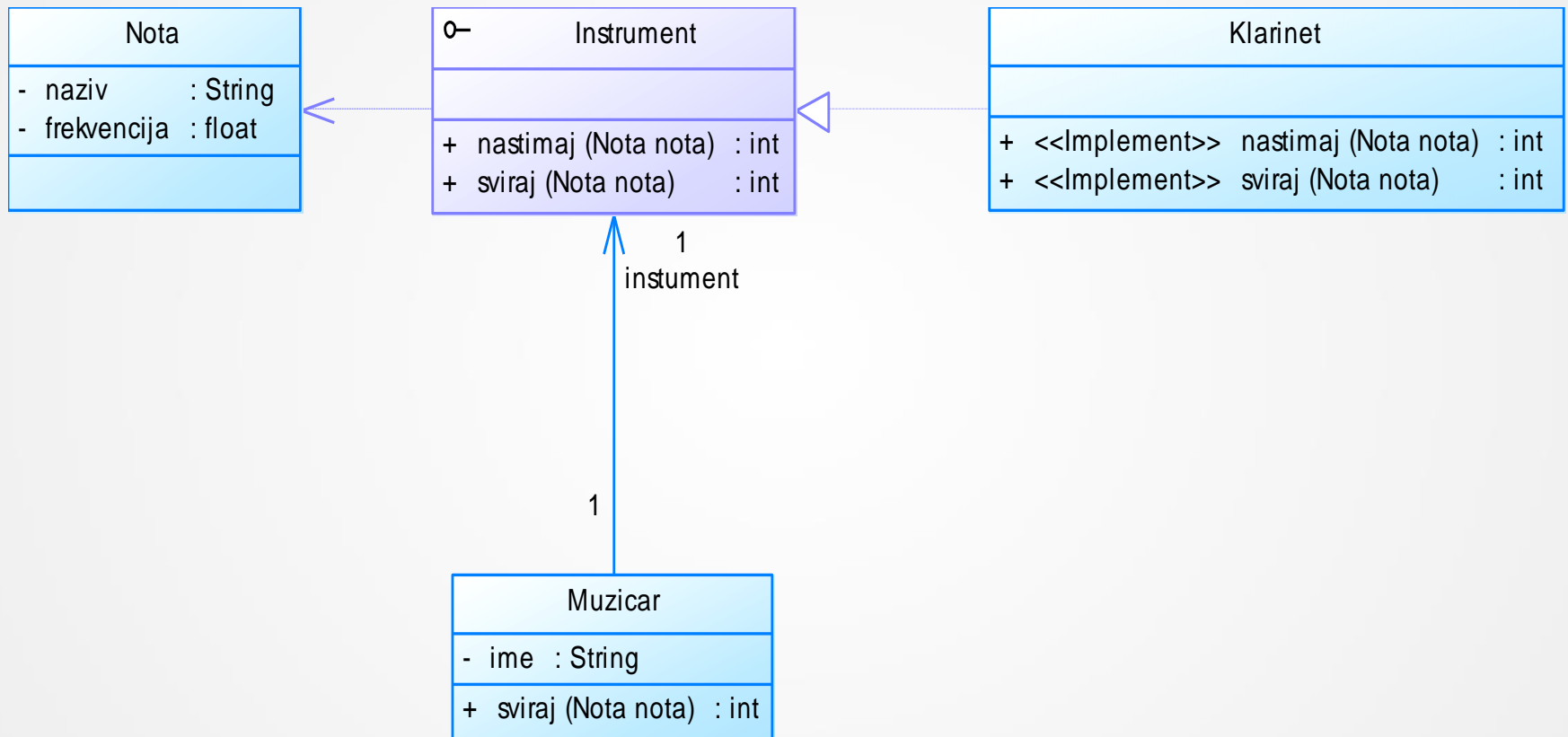
- ☪ Omogućavaju definisanje samo apstraktnih metoda, konstanti i statičkih atributa
- ☪ Ključna reč **implements**
- ☪ **Interfejs nije klasa! On je spisak metoda i atributa koje klasa koja implementira interfejs mora da poseduje.**
- ☪ **Predstavlja neku vrsta ugovora kojom se obavezuje klasa koja ga implementira da će podržati određeno ponašanje.**
- ☪ Interfejsi se ne nasleđuju, već implementiraju
- ☪ Da bi klasa implementirala interfejs, mora da redefiniše sve njegove metode
- ☪ Jedan interfejs može da nasledi jedan, ili više interfejsa
- ☪ Sve metode su implicitno **public**, a svi atributi su implicitno **public static final**

Interfejsi

```
public class Racunar {  
    public HardDisk hardDisk;  
    public int upali() {  
    }  
}  
public interface HardDisk {  
    int pomeriGlavu();  
}  
public class SATAHardDisk implements HardDisk {  
    @Override  
    public int pomeriGlavu() {  
        ...  
    }  
}
```



Interfejsi



Interfejsi

```
interface Instrument {
    int sviraj(Nota nota);
    int nastimaj(Nota nota);
}

class Klarinet implements Instrument {
    @Override
    public int sviraj(Nota nota) { ... }
    @Override
    public int nastimaj(Nota nota) { ... }
}

class Muzicar {
    Instrument instrument;
    int sviraj(Nota nota) {
        return instrument.sviraj(nota);
    }
}

...
Muzicar m = new Muzicar();
m.instrument = new Klarinet();
m.sviraj(nota);
```

Interfejsi

- ☕ Jedna klasa može da implementira jedan ... ili više interfejsa

```
interface USB {  
    void init();  
    byte[] getData();  
}  
  
interface Camera {  
    void init();  
    Picture getPicture();  
}  
  
class WebCam implements USB, Camera {  
    @Override  
    void init() { ... }  
    @Override  
    byte[] getData() { ... }  
    @Override  
    Picture getPicture() { ... }  
}
```

primer 04

Interfejsi – default metoda

☞ Interfejs može da sadrži default metode

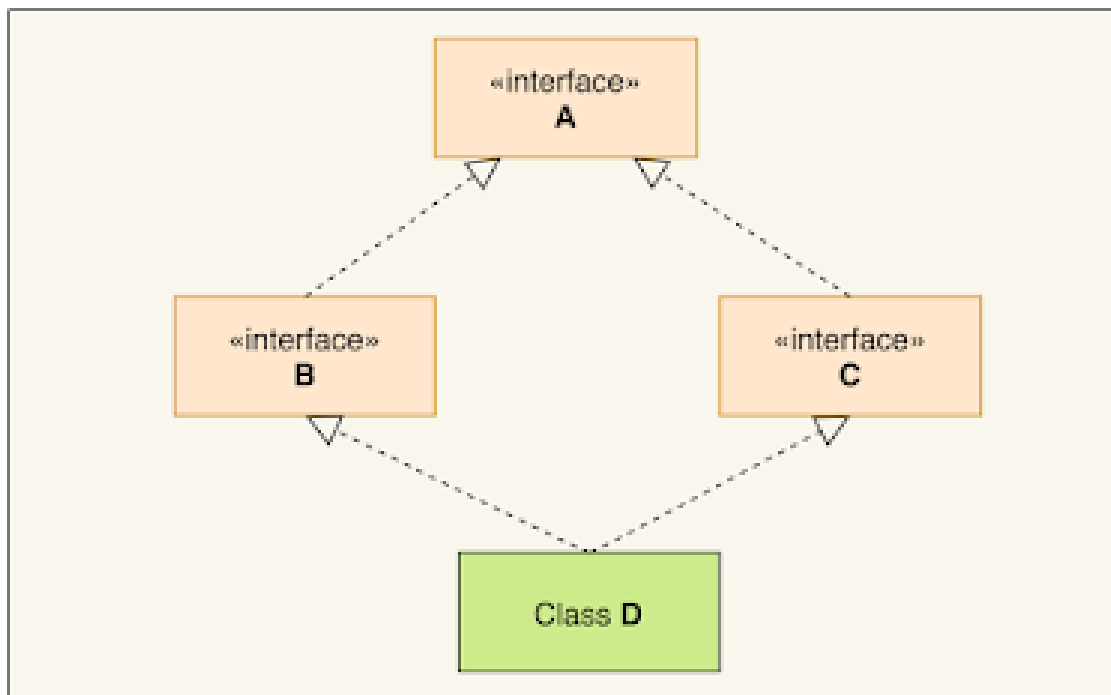
```
public class Racunar {
    public HardDisk hardDisk;
    public int upali() {
        hardDisk.pomeriGlavu();
    }
    public int ocitaj() {
        hardDisk.proveriSistem();
    }
}

public interface HardDisk {
    default int proverisistem()
        System.out.println("Provera rada sistema!");
        System.out.println("Uspešno! ");
        System.out.println("Startovanje!");
        return pomeriGlavu();
    }
    int pomeriGlavu();
}

public class SATAHardDisk implements HardDisk {
    @Override
    public int pomeriGlavu() {
        ...
    }
}
```


Interfejsi višestruko nasleđivanje

- ☞ Jedna interfejs može da nasledi jedan ... ili više interfejsa
- ☞ Nije problema ako su samo definicije meetoda
- ☞ Problem default metode
- ☞ Isto važi i za implementaciju više interfejsa u jednoj klasi



Interfejsi problem za višestruko nasleđivanje default metoda

```
public interface Vehicle {  
    default void print() {  
        System.out.println("I am a vehicle!");  
    }  
}  
  
public interface FourWheeler {  
    default void print() {  
        System.out.println("I am a four wheeler!");  
    }  
}  
  
public class Car implements Vehicle, FourWheeler {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.print();  
    }  
}
```

Interfejsi rešenje za višestruko nasleđivanje default metoda

```
public interface Vehicle {  
    default void print() {  
        System.out.println("I am a vehicle!");  
    }  
}  
  
public interface FourWheeler {  
    default void print() {  
        System.out.println("I am a four wheeler!");  
    }  
}  
  
public class Car implements Vehicle, FourWheeler {  
    public void print() {  
        System.out.println("I am a four wheeler car  
vehicle!");  
    }  
}
```

primer 04.višestruko

Inner classes (unutrašnje klase)

```
class Spoljasnja {  
    Spoljasnja() { ... }  
    void metoda() { ... }  
  
    class Unutrasnja {  
        void metoda() { ... }  
    }  
}
```

☞ Konstrukcija objekta unutrašnje klase izvan spoljašnje klase

```
Spoljasnja sp = new Spoljasnja();  
Spoljasnja.Unutrasnja un = sp.new Unutrasnja();
```

Statičke unutrašnje klase






```
class Spoljasnja {  
    void metoda() { ... }  
    static class UnutrasnjaStatic {  
        int metoda2() { ... }  
    }  
}  
  
...  
Spoljasnja.UnutrasnjaStatic u =  
new Spoljasnja.UnutrasnjaStatic();
```

Statičko povezivanje (*static/early binding*) i dinamičko povezivanje (*dynamic/late binding*)

- ☞ Povezivanje poziva funkcije sa odgovarajućim telom metode (implementacijom) se naziva *binding*.
- ☞ Direktno je povezano sa polimorfizmom za vreme kompajliranja i polimorfizmom za vreme izvršavanja.
- ☞ Povezivanje se odnosi na određivanje konkretnih vrednosti promenljivih i određivanje konkretnih implementacija metoda.

Statičko povezivanje (*static/early binding*) i dinamičko povezivanje (*dynamic/late binding*)

Static binding

-  kada je tip objekta određen za vreme kompajliranja
-  kada je poziv metode određen za vreme kompajliranja (konkretna implementacija metoda zna za vreme kompajliranja).
 -  Koristi informacije o tipu objekta
 -  Sve metode deklarisanе kao static, private ili final se sigurno određene za vreme kompajliranja (ne mogu da se redefinišu).
-  Statičko povezivanje odnosi se slučaj kada se zna koja će konkretna implementacija metoda biti pozvana još za vreme kompajliranja (pogledati primer polimorfizma za vreme kompajliranja).

Statičko povezivanje (*static/early binding*) i dinamičko (*dynamic/late binding*) povezivanje







Static binding

```
public class Test {
    public class Vozilo {
        public void pokreni(){System.out.println("Vozilo je upaljeno");}
    }
    public class Automobil extends Vozilo {
        @Override
        public void pokreni(){System.out.println("Automobil je upaljen");}
    }
    void ispisNesto(int a){System.out.println("Ispis parametar tipa int");}
    void ispisNesto(double d){System.out.println("Ispis parametar tipa double");}
    void ispisNesto(Vozilo v){System.out.println("Ispis parametar Vozilo");}
    void ispisNesto(Automobil a){System.out.println("Ispis parametar Automobil");}

    public void staticBindingTest() {
        System.out.println("Poziv konkretne metode zavisi od tipa parametra");
        ispisNesto(4); //Ispis parametar tipa int
        ispisNesto(5.0); //Ispis parametar tipa double
        ispisNesto(new Vozilo()); //Ispis parametar tipa Vozilo
        ispisNesto(new Automobil()); //Ispis parametar tipa Automobil
        Vozilo v = new Automobil();
        ispisNesto(v); //Ispis parametar tipa Vozilo
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.staticBindingTest();
    }
}
```


Statičko povezivanje (*static/early binding*) i dinamičko (*dynamic/late binding*) povezivanje

Dynamic binding

-  kada se konkretan tip objekta određuje za vreme izvršavanja
-  kada je poziv metode određuje za vreme izvršavanja (konkretna implementacija metoda nije poznata za vreme kompajliranja i određuje se za vreme izvršavanja, pogledati primer).
 -  koristi konkretne objekte
-  Dinamičko povezivanje odnosi se na slučaj kada konkretna implementacija metoda nije poznata za vreme kompajliranja i određuje se za vreme izvršavanja (pogledati primer polimorfizma za vreme izvršavanja)

Statičko povezivanje (*static/early binding*) i dinamičko (*dynamic/late binding*) povezivanje

Dynamic binding

```
public class Test {
    public class Vozilo {
        public void pokreni(){System.out.println("Vozilo je upaljeno");}
    }
    public class Automobil extends Vozilo {
        @Override
        public void pokreni(){System.out.println("Automobil je upaljen");}
    }
    public void dynamicBindingTest() {
        System.out.println("Poziv konkretne metode zavisi od objekta koji poziva metodu");
        Vozilo v = new Automobil();
        v.pokreni(); //Automobil je upaljen
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.dynamicBindingTest();
    }
}
```