Sveznadar

MKE

Obradio i prikupio
DuškoMilinčić

JAVA

OOP u Javi

BL, 2003

Koncept objektno-orjentisanog programiranja kod Jave

1. Osnovni koncepti i prednosti OOP u Javi

Ovdje ćemo govoriti o osnovnim konceptima OOP. Objektno-orijentisano programiranje je paradigma koja sadrži mnoge koncepte kao što su **nasljeđivanje (inheritance)**, **povezivanje podataka (data binding)**, **polimorfizam** itd.
Prvi objektno-orijentisani programski jezici su bili **Simula** i **Smalltalk**. Programska paradigma gdje je sve predstavljeno kao objekt je poznata kao istinski objektno-orijentisani programski jezik.

1.1 Objektno-orijentisani sistem programiranja

Objekt označava entitet iz stvarnog svijeta kao što su olovka, stolica, stol itd. **Objektno-orijentisano programiranje** je metodologija ili paradigma pisanja programa koristeći klase i objekte. Ono pojednostavljuje razvoj i održavanje softvera time što obezbjeđuje sljedeće koncepte:

- Objekt
- Klasa
- Nasljeđivanje
- Polimorfizam
- Apstrakcija
- Enkapsulacija

Objekt
Bilo koji entitet koji ima stanje i ponašanje se naziva objekt. Napr.: stolica, olovka, stol, tastatura, bicikl itd. Može biti fizički i logički.

Klasa
Kolekcija objekata se naziva klasa. Ona je logički entitet.

Nasljeđivanje
Kada jedan objekt dobija sve osobine i ponašanja roditeljskog objekta to se naziva nasljeđivanje. Ono obezbjeđuje ponovnu upotrebljivost (reusability) koda. Koristi se da se postigne polimorfizam u vremenu izvršavanja.

Polimorfizam
Kada se **jedan zadatak izvršava na različite načine** to je poznato kao polimorfizam. Napr.: uvjeriti mušteriju da nešto kupi možemo na različite načine, nacrtati nešto možemo na razne načine itd.
U Javi, koristimo preopterećenje (overloading) metoda i preklapanje (overriding) metoda da bismo postigli polimorfizam.

Apstrakcija
Sakrivanje unutrašnjih detalja i pokazivanje funkcionalnosti je poznato kao apstrakcija.
Napr.: dok telefoniramo nije neophodno da znamo kako se procesira signal.
U Javi, koristimo apstraktne klase i interfejs da postignemo apstrakciju.

Enkapsulacija
Povezivanje (ili uvezivanje) koda i podataka zajedno u jedinstvenu jedinicu je poznato kao enkapsulacija.
Java klasa je primjer enkapsulacije. Java zrno (bean) je u potpunosti enkapsulirana klasa zato što su svi podaci-članovi u njoj privatni.

1.1.2 Prednosti OOP nad proceduralno-orijentisanim programskim jezicima

1) OOP čini razvoj i održavanje lakšim dok u proceduralno-orijentisanim programskim jezicima to nije lako postići ukoliko kod raste kako raste obim projekta.

2) OOP obezbjeđuje sakrivanje podataka dok se u proceduralno-orijentisanim programskim jezicima globalnim podacima može pristupiti sa bilo kog mjesta.

3) OOP daje mnogo efikasnije mogućnosti simuliranja događaja iz realnog svijeta. Možemo obezbijediti rješavanje realnih problema ako koristimo OOP jezik.

Razlika između objektno-orijentisanog i objektno-zasnovanog programskog jezika

Objektno zasnovani programski jezik slijedi sve principe OOP osim nasljeđivanja. JavaScript i VBScript su primjeri objektno zasnovanih programskih jezika.

2. Konvencije o imenovanjima u Javi

Konvencije za imenovanja u Javi su pravila koja treba slijediti prilikom odlučivanja koje ime dati identifikatorima kao što su klasa, paket, varijabla, konstanta, metoda itd.

Sve klase, interfejsi, paketi, metodi i polja u programskom jeziku Java se imenuju u skladu s tom konvencijom.

2.1.1 Prednosti konvencije o imenovanjima u Javi

Koristeći standardne konvencije o imenovanjima u Javi, kod postaje lakši za čitanje kako za onoga ko ga je pisao tako i za druge programere. Čitljivost Java programa je veoma važna. Ona znači da je potrebno **manje vremena** da se shvati šta kod radi.

Ime	Konvencija
Ime klase	Treba početi velikim slovom i biti imenica napr. String, Color, Button, System, Thread itd.
Ime interfejsa	Treba početi velikim slovom i biti pridjev napr. Runnable, Remote, ActionListener itd.
Ime metode	Treba početi malim slovom i biti glagol napr. actionPerformed(), main(), print(), println() itd.
Ime varijable	Treba početi malim slovom napr. firstName, orderNumber itd.
Ime paketa	Treba biti ispisano malim slovima napr. java, lang, sql, util itd.
Ime konstante	Treba biti ispisano velikim slovima napr. RED, YELLOW, MAX_PRIORITY itd.

2.1.2 CamelCase u konvencijama o imenovanjima u Javi

Java slijedi tzv. camelcase sintaksu za imenovanje klase, interfejsa, metode i varijable.
Ako je ime kombinovano od dvije riječi, druga riječ će uvijek početi sa velikim slovom napr. actionPerformed(), firstName, ActionEvent, ActionListener itd.

3. Objekt i klasa u Javi

U objektno-orijentisanim programskim tehnikama, program se piše koristeći objekte i klase. Objekt može biti kako fizički tako i logički entitet dok je klasa samo logički entitet.

3.,1.1 Objekt u Javi

Entitet koji ima stanje i ponašanje se naziva objekt napr. stolica, bicikl, marker, olovka, stol, auto itd. On može biti fizički ili logički. Primjer logičkog objekta je bankarski sistem.
Objekt ima tri karakteristike:

- **stanje:** predstavlja podatke (vrijednosti)objekta.
- **ponašanje:** predstavlja ponašanje (funkcionalnost) objekta kao što su depozit, podizanje novca itd.
- **identitet:** identitet objekta se tipično implementira preko jedinstvenog ID. Vrijednost ID nije vidljiva vanjskom korisniku. Ali, ona se koristi interno od strane JVM (Java Virtual Machine) da identifikuje svaki objekt na jedinstven način.

Napr. : Olovka je objekt. Njeno ime (marka) je napr. Centropen, boja je bijela itd. što čini njeno stanje. Koristi se za pisanje, tako da je pisanje njeno ponašanje.

Objekt je instanca klase. Klasa je šablon ili nacrt po kojem se kreiraju objekti. Znači, objekt je instanca (rezultat) klase.

3.,1.2 Klasa u Javi

Klasa je grupa objekata koji imaju zajedničke osobine. Ona je šablon ili nacrt po kojem se objekti kreiraju.
Klasa u Javi može sadržati:

- **podatak-član**
- **metodu**
- **konstruktor**
- **blok**
- **klasu i interfejs**

Sintaksa za deklarisanje klase

```
class <class_name>{
   data member;
   method;
}
```

3.1.2 Jednostavan primjer objekta i klase

U ovom primjeru, kreiraćemo klasu Student koja ima dva podatka člana id i ime. Kreiraćemo objekt klase Student pomoću ključne riječi new i ispisaćemo vrijednost objekta.

```
class Student1{
 int id; //data member (podatak-član, također instansna varijabla)
 String name; //data member(podatak-član, također instansna varijabla)

 public static void main(String args[]){
  Student1 s1=new Student1(); //kreiranje objekta klase Student
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

```
Izlaz:0 null
```

3.1.3 Instansna varijabla u Javi

Varijabla koja je kreirana unutar klase ali izvan metoda, se naziva instansna varijabla. Instansna varijabla ne dobija memoriju u vremenu kompajliranja. Ona dobija memoriju u vremenu izvršavanja kada je objekt (instanca) kreiran. To je razlog zbog kojeg se naziva instansna varijabla.

3.1.4 Metoda u Javi

U Javi, metoda je poput funkcije tj. koristi se da izrazi ponašanje objekta.

Prednosti metode

- Ponovna upotrebljivost koda (reusability)
- Optimizacija koda

Ključna riječ new

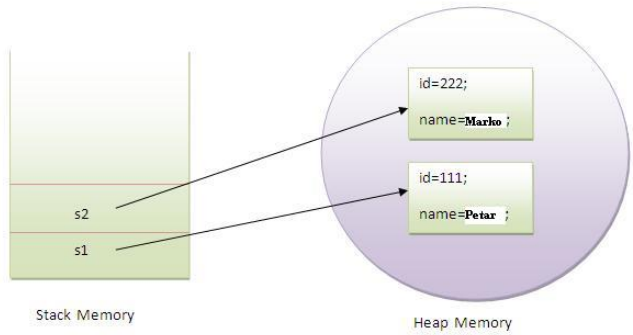
Ključna riječ new se koristi da alocira (dodijeli) memoriju u vremenu izvršavanja.

3.1.5 Primjer objekta i klase koji sadrže zapise o studentima

U ovom primjeru, kreiraćemo dva objekta klase Student i inicijalizirаćemo vrijednost ovih objekata pozivajući metod insertRecord. Pokazaćemo stanje (podatke) objekata pozivajući metod displayInformation.

```
class Student2{
 int rollno;
 String name;

 void insertRecord(int r, String n){  //metod
  rollno=r;
  name=n;
 }

 void displayInformation(){System.out.println(rollno+" "+name);} //metod

 public static void main(String args[]){
  Student2 s1=new Student2();
  Student2 s2=new Student2();

  s1.insertRecord(111,"Petar");
  s2.insertRecord(222,"Marko");

  s1.displayInformation();
  s2.displayInformation();

 }
}
```

```
Izlaz: 111 Petar
       222 Marko
```



Kao što se može vidjeti na slici, objekt dobija memoriju u Heap oblasti a referentne (adresne) varijable upućuju na objekt alociran u Heap memorijskoj oblasti. Ovdje su i s1 i s2 referentne varijable koje upućuju na objekte alocirane u memoriji.

3.1.6 Drugi primjer objekta i klase

Ovdje je dat još jedan primjer koji sadrži zapise o klasi Rectangle. Objašnjenje je isto kao u prethodnom primjeru za klasu Student.

```
class Rectangle{
 int length;
 int width;

 void insert(int l,int w){
  length=l;
  width=w;
 }

 void calculateArea(){System.out.println(length*width);}

 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();

  r1.insert(11,5);
  r2.insert(3,15);

  r1.calculateArea();
  r2.calculateArea();
}
}
```

```
Izlaz: 55
       45
```

3.1.6 Različiti načini za kreiranje objekta u Javi

Postoji više načina za kreiranje objekta u Javi. To su:

- pomoću ključne riječi new
- pomoću metode newInstance()
- pomoću metode clone()
- pomoću factory metode itd.

3.1.7 Anonimni objekt

Anonimni jednostavno znači bez imena. Objekt koji nema reference se naziva anonimni objekt. Ako je potrebno neki objekt upotrijebiti samo jednom, anonimni objekt je dobar pristup.

```
class Calculation{
   void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("faktorijel je "+fact);
}
  public static void main(String args[]){
 new Calculation().fact(5); //pozivanje metoda sa anonimnim objektom
}
}
Izlaz:Faktorijel je 120
```

3.1.8 Kreiranje više objekata pomoću samo jednog tipa

Moguće je kreirati više objekata pomoću samo jednog tipa kao što se radi u slučaju primitiva.

```
    Rectangle r1=new Rectangle(),  r2=new Rectangle();  //kreiranje dva objekta
```

Pogledajmo primjer:

```
class Rectangle{
 int length;
 int width;

 void insert(int l,int w){
  length=l;
  width=w;
 }
   void calculateArea(){System.out.println(length*width);}
   public static void main(String args[]){
  Rectangle r1=new Rectangle(),  r2=new Rectangle();  //kreiranje dva objekta

  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}

Izlaz: 55
       45
```

4. Preopterećenje (overloading) metoda u Javi

Ako klasa ima više metoda istog imena ali različitih parametara, to je poznato kao

Preopterećenje (overloading) metoda.

Ako se mora izvesti samo jedna operacija, isto ime metoda poboljšava čitljivost programa. Pretpostavimo da treba izvesti sabiranje datih brojeva ali da može biti bilo koji broj argumenata, ako napišemo metod kao što je a(int,int) za dva parametra, i b(int,int,int) za tri parametra tada može biti teško kako autoru tako i drugim programerima da shvate ponašanje metoda zato što im se imena razlikuju. Zato uvodimo preopterećenje metoda da bismo brže razumjeli program.

Prednosti preopterećenja metoda
Preopterećenje metoda **poboljšava čitljivost programa**.

Različiti načini preopterećenja metode
Postoje dva načina preopterećenja metoda u Javi

1. Promjena broja argumenata
2. Promjena tipa podataka

Napomena: U Javi, preopterećenje metoda nije moguće promjenom return tipa metode (tipa koji metoda vraća).

4.1 Primjer preopterećenja metode promjenom broja argumenata

U ovom primjeru, kreiraćemo dve preopterećene metode, prva sum metoda vrši sabiranje dva broja, a druga sum metoda vrši sabiranje tri broja.

```
class Calculation{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  Calculation obj=new Calculation();
  obj.sum(10,10,10);
  obj.sum(20,20);

  }
}
```

```
Izlaz: 30
       40
```

4.2 Primjer preopterećenja metode promjenom tipa podataka argumenata

U ovom primjeru, kreiraćemo dva preopterećena metoda koji se razlikuju po tipu podataka. Prva sum metoda prima dva cjelobrojna (int) argumenta, a druga sum metoda prima dva double argumenta.

```
class Calculation2{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(double a,double b){System.out.println(a+b);}

  public static void main(String args[]){
  Calculation2 obj=new Calculation2();
  obj.sum(10.5,10.5);
  obj.sum(20,20);

  }
}
```

```
Izlaz: 21.0
       40
```

4.2.1 Zašto nije moguće preopterećenje metode promjenom return tipa metode?

U Javi, preopterećenje metode nije moguće promjenom return tipa metode zato što se može javiti dvosmislenost. Pogledajmo kako do nje može doći. Javiće se ovakav problem:

```
class Calculation3{
  int sum(int a,int b){System.out.println(a+b);}
  double sum(int a,int b){System.out.println(a+b);}

  public static void main(String args[]){
  Calculation3 obj=new Calculation3();
  int result=obj.sum(20,20); //Compile Time Error  (greška u vremenu kompajliranja)

  }
}
```

```
Izlaz: 65.54/Calculation3.java:3: error: method sum(int,int) is already defined in class Calculation3
ntln(a+b);}
```

4.2.2 Može li se preopteretiti main() metoda?

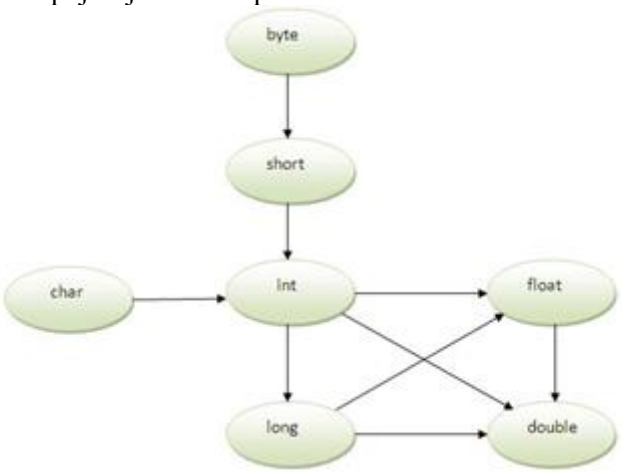
Da, može. Moguće je imati bilo koji broj main metoda u klasi pomoću preopterećenja metode. Pogledajmo jednostavan primjer:

```
class Overloading1{
  public static void main(int a){
  System.out.println(a);
  }

  public static void main(String args[]){
  System.out.println("main() metoda pozvana");
  main(10);
  }
}
```

```
Izlaz: main() metoda pozvana
       10
```

4.2..3 Preopterećenje metode i promocija tipa (TypePromotion)

Jedan tip je promoviran u drugi implicitno ako nije pronađen nijedan podudarajući tip podatka. Sljedeća slika pojašnjava koncept:



Kao što se vidi na dijagramu, byte se može promovisati u short, int, long, float ili double. Tip podatka short može se promovisati u int, long, float ili double. Tip char se može promovisati u int, long, float ili double itd.

Primjer preopterećenja metoda sa promocijom tipa

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

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

```
 OverloadingCalculation1 obj=new OverloadingCalculation1();
 obj.sum(20,20); //ovdje će drugi int literal biti promovisan u long
 obj.sum(20,20,20);
 }
}

Izlaz: 40
       60
```

4.2.3 Primjer preopterećenja metoda sa promocijom tipa ako je pronađeno poklapanje

Ako postoji poklapanje tipova argumenata u metodi, promocija tipa se ne vrši.

```
class OverloadingCalculation2{
 void sum(int a,int b){System.out.println("int arg metod pozvan");}
 void sum(long a,long b){System.out.println("long arg metod pozvan");}

 public static void main(String args[]){
 OverloadingCalculation2 obj=new OverloadingCalculation2();
 obj.sum(20,20); //ovdje je int arg sum() metod pozvan
 }
}

Izlaz:int arg metod pozvan
```

4.2.4 Primjer preopterećenja metoda sa promocijom tipa u slučaju dvosmislenosti

Ako u metodi nema argumenata koji se slažu po tipu, i svaki metod promoviše sličan broj argumenata, doći će do dvosmislenosti.

```
class OverloadingCalculation3{
 void sum(int a,long b){System.out.println("a metod pozvan");}
 void sum(long a,int b){System.out.println("b metod pozvan");}

 public static void main(String args[]){
 OverloadingCalculation3 obj=new OverloadingCalculation3();
 obj.sum(20,20); //ovdje postoji dvosmislenost
 }
}

Izlaz:Compile Time Error
```

Napomena: Tip se ne de-promoviše implicitno, napr. double se ne može depromovisati u neki drugi tip implicitno.

5. Konstruktor u Javi

Konstruktor je **specijalni tip metode** koji se koristi da inicijalizuje objekt. Konstruktor se poziva u trenutku kreiranja objekta. On konstruiše vrijednosti, tj. obezbjeđuje podatke za objekt i otuda mu to ime.

Pravila za kreiranje konstruktora

U osnovi, postoje dva definisana pravila za konstruktore:

1. Ime konstruktora mora biti isto kao ime njegove klase
2. Konstruktor ne smije vraćati nikakav određeni tip (čak ni **void**)

Tipovi konstruktora

Postoje dva tipa konstruktora:

1. Podrazumijevani (default, no-arg) konstruktor
2. Parametrizovani konstruktor

1) Podrazumijevani konstruktor

Ovaj konstruktor nema parametara, a njegova sintaksa je:

<ime_klase>(){}

Primjer 1:

Kreiranje podrazumijevanog konstruktora u klasi Bike. On će biti pozvan u trenutku kreiranja objekta.

```
class Bike1{
Bike1(){System.out.println("Bike je kreiran");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
Izlaz: Bike je kreiran
```

Pravilo: Ako nema konstruktora u klasi, kompajler automatski kreira podrazumijevani konstruktor.

Svrha postojanja podrazumijevanog konstruktora je da obezbijedi podrazumijevane (default) vrijednosti za objekt, napr. 0, null itd., u zavisnosti od tipa.

Primjer 2:

Podrazumijevani konstruktor koji prikazuje podrazumijevane vrijednosti.

```
class Student3{
int id;
String ime;

void display(){System.out.println(id+" "+ime);}
public static void main(String args[]){
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
}
}
```

```
Izlaz: 0 null
       0 null
```

*Objašnjenje***:** U ovoj klasi nije kreiran nikakav konstruktor, tako da kompajler obezbjeđuje podrazumijevani konstruktor.
Vrijednosti 0 i null je obezbijedio podrazumijevani konstruktor.

5.1.1 Parametrizovani konstruktor

Ovaj konstruktor ima parametre i koristi se da obezbijedi različite vrijednosti za zasebne objekte.
Primjer 3:
Kreiranje konstruktora klase Student koji ima dva parametra. Moguće je imati bilo koji broj parametara u konstruktoru.

```
class Student4{
   int id;
   String ime;

   Student4(int i,String n){
   id = i;
   ime = n;
   }
   void display(){System.out.println(id+" "+ime);}

   public static void main(String args[]){
   Student4 s1 = new Student4(111,"Petar");
   Student4 s2 = new Student4(222,"Marko");
   s1.display();
   s2.display();
  }
}
```

```
Izlaz: 111 Petar
       222 Marko
```

5.1.2 Preopterećenje (overloading) konstruktora

Ovo je tehnika u Javi u kojoj klasa može imati bilo koji broj konstruktora koji se razlikuju u parametarskim listama. Kompajler pravi razliku između ovih konstruktora tako što uzima u obzir broj parametara u listi i njihov tip.

Primjer 4:
Preopterećenje konstruktora

```
class Student5{
   int id;
   String ime;
   int starost;
```

```
    Student5(int i,String n){
    id = i;
    ime = n;
    }
    Student5(int i,String n,int a){
    id = i;
    ime = n;
    starost=a;
    }
    void display(){System.out.println(id+" "+ime+" "+starost);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Petar ");
    Student5 s2 = new Student5(222,"Marko",25);
    s1.display();
    s2.display();
   }
}
```

```
Izlaz: 111 Petar 0
       222 Marko 25
```

5.1.3 Razlika između konstruktora i metode

Konstruktor	Metoda
Konstruktor se koristi da inicijalizuje stanje objekta.	Metoda se koristi da izrazi ponašanje objekta.
Konstruktor ne smije imati return tip.	Metoda mora imati return tip.
Konstruktor se poziva implicitno.	Metoda se poziva eksplicitno.
Java kompajler obezbjeđuje podrazumijevani konstruktor ako nema konstruktora.	Metodu ne obezbjeđuje kompajler ni u kom slučaju.
Ime konstruktora mora biti isto kao ime klase.	Ime metode može ali i ne mora biti isto kao ime klase.

5.1.4 Kopiranje vrijednosti jednog objekta u drugi (kao konstruktor kopije u C++)

Postoji mnogo načina za kopiranje vrijednosti jednog objekta u drugi. Neki od njih su:

- Pomoću konstruktora
- Dodjeljivanjem vrijednosti jednog objekta drugom
- Pomoću metode clone() iz klase Object

Primjer 5:
Kopiranje vrijednosti jednog objekta u drugi pomoću konstruktora.

```
class Student6{
   int id;
   String ime;
   Student6(int i,String n){
   id = i;
   ime = n;
   }

   Student6(Student s){
   id = s.id;
   ime =s.ime;
   }
   void display(){System.out.println(id+" "+ime);}

   public static void main(String args[]){
   Student6 s1 = new Student6(111,"Petar");
   Student6 s2 = new Student6(s1);
   s1.display();
   s2.display();
  }
}
```

```
Izlaz: 111 Petar
       111 Petar
```

Primjer 6:

Kopiranje vrijednosti jednog objekta u drugi dodjeljivanjem vrijednosti jednog objekta drugom . U ovom slučaju nema potrebe za kreiranjem konstruktora.

```
class Student7{
   int id;
   String ime;
   Student7(int i,String n){
   id = i;
   ime = n;
   }
```

```
    Student7(){}
    void display(){System.out.println(id+" "+ime);}

    public static void main(String args[]){
    Student7 s1 = new Student7(111,"Petar");
    Student7 s2 = new Student7();
    s2.id=s1.id;
    s2.ime=s1.ime;
    s1.display();
    s2.display();
   }
}
```

```
Izlaz: 111 Petar
       111 Petar
```

Napomena: Konstruktor može izvršiti i druge zadatke osim inicijalizacije, kao što su kreiranje objekta, pozivanje metode itd. Bilo koja operacija se može izvršiti u konstruktoru kao i u metodi.

6. Ključna riječ static

Ključna riječ static se koristi u Javi uglavnom za upravljanje memorijom. Može se primijeniti na varijable, metode, blokove i ugniježdene klase. Ključna riječ static pripada više klasi nego instanci klase.
Statički mogu biti:

1. Varijabla (zove se još i klasna varijabla)
2. Metoda (zove se još i klasna metoda)
3. Blok
4. Ugniježdena klasa

6.1.1 Statička varijabla

Varijabla deklarisana pomoću ključne riječi static se naziva statička varijabla.

- Statička varijabla se može upotrijebiti tako da se odnosi na zajedničke osobine svih objekata (onih koje nisu jedinstvene za svaki objekt), napr. kompanija – imena zaposlenih, fakultet – imena studenata, itd.
- Statičkoj varijabli se dodjeljuje memorija samo jednom u prostoru klase u trenutku učitavanja klase.

Prednosti statičke varijable
Statička varijabla čini program memorijski efikasnijim (tj. štedi memoriju).
Razumijevanje problema bez statičke varijable

```
class Student{
    int brind;
    String ime;
    String fakultet="ITS";
}
```

Pretpostavimo da na fakultetu ima 500 studenata, pa će sve instansne varijable dobiti memoriju svaki put kada se objekt kreira. Svaki student ima svoj jedinstveni broj indeksa (brind) i ime tako da je instansna varijabla dobra. Ovdje fakultet označava zajedničku osobinu svih objekata. Ako je učinimo statičkom, ovo će polje dobiti memoriju samo jednom.

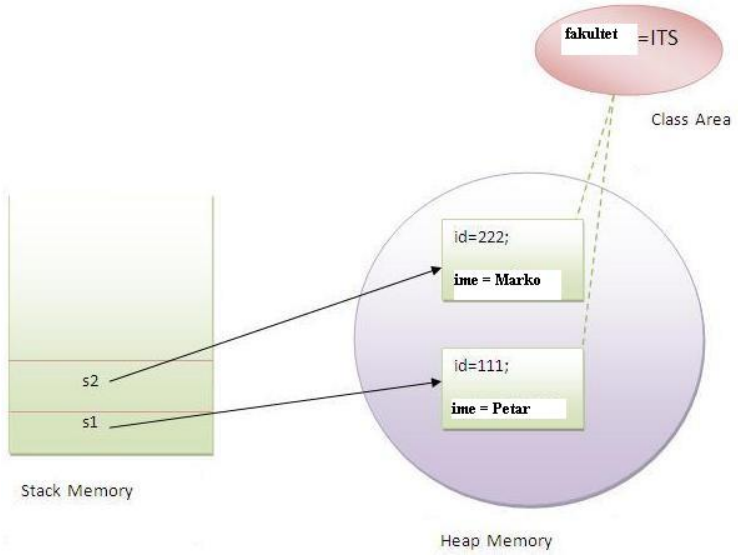
Napomena1: Statička osobina se dijeli na sve objekte.

Primjer1:

```
  //Program sa statičkom varijablom

   class Student8{
     int brind;
     String ime;
     static String fakultet ="ITS";

     Student8(int r,String n){
```

```
    brind = r;
    ime = n;
    }
   void display (){System.out.println(brind+" "+ime+" "+fakultet);}

   public static void main(String args[]){
   Student8  s1 = new Student8(111,"Petar");
   Student8  s2 = new Student8(222,"Marko");

   s1.display();
   s2.display();
   }
  }
```

```
Izlaz: 111 Petar ITS
       222 Marko ITS
```



Primjer 2:
Program brojač (counter) bez statičke varijable.

U ovom primjeru kreiraćemo instansnu varijablu imena count koja se inkrementira u konstruktoru. Pošto instansna varijabla dobija memoriju u trenutku kreiranja objekta, svaki objekt će imati kopiju instansne varijable, pa ako se inkrementira to se neće odraziti na druge objekte. Tako će svaki objekt imati vrijednost 1 u varijabli count.

```
class Counter{
int count=0;          //dobija memoriju kada se instanca kreira

Counter(){
count++;
System.out.println(count);
```

```
}

public static void main(String args[]){

Counter  c1=new Counter();
Counter  c2=new Counter();
Counter  c3=new Counter();

 }
}
```

```
Izlaz: 1
       1
       1
```

Primjer 3:
Program brojač (counter) sa statičkom varijablom.

Kao što je već rečeno, statička varijabla će dobiti memoriju samo jednom, pa ako bilo koji objekt promijeni vrijednost statičke varijable, ona će zadržati tu vrijednost.

```
class Counter{
static int count=0;          //dobija memoriju samo jednom i zadržava svoju vrijednost

Counter(){
count++;
System.out.println(count);
}

public static void main(String args[]){

Counter  c1=new Counter();
Counter  c2=new Counter();
Counter  c3=new Counter();

 }
}
```

```
Izlaz: 1
       2
       3
```

6.1.2 Statička metoda

Ako primijenimo ključnu riječ static na bilo koji metod, on postaje statički metod

- statički metod pripada klasi a ne objektu klase.
- statički metod može biti pozvan bez potrebe da se kreira instanca klase.
- statički metod može pristupiti statičkim podacima-članovima i može mijenjati njihovu vrijednost.

Primjer statičke metode

```
//Program koji mijenja zajedničke osobine svih objekata (statičko polje).

class Student9{
    int rollno;
    String name;
    static String college = "ITS";

    static void change(){
    college = "BBDIT";
    }

    Student9(int r, String n){
    rollno = r;
    name = n;
    }

    void display (){System.out.println(rollno+" "+name+" "+college);}

   public static void main(String args[]){
   Student9.change();

   Student9 s1 = new Student9 (111,"Petar");
   Student9 s2 = new Student9 (222,"Marko");
   Student9 s3 = new Student9 (333,"Janko");

   s1.display();
   s2.display();
   s3.display();
   }
}


Izlaz: 111 Petar BBDIT
       222 Marko BBDIT
       333 Janko BBDIT
```

Drugi primjer statičke metode koja vrši normalne kalkulacije

```
//Program koji računa kub datog broja pomoću statičke metode

class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

```
Izlaz:125
```

Ograničenja statičkog metoda

Postoje dva glavna ograničenja za statički metod. To su:

1. statički metod ne može koristiti ne-statičke podatke-članove ili pozvati ne-statički metod direktno.
2. this i super se ne mogu koristiti u statičkom kontekstu.

```
class A{
 int a=40; //ne-statički

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

```
Izlaz:Compile Time Error
```

Zašto je main metod statičan?

Zato što objekt ne mora da poziva statički metod; ako bi main bio ne-statički metod, JVM bi prvo kreirao objekt a zatim pozvao main() metod što bi dovelo do problema alokacije dodatne memorije.

6.1.3 Statički blok

- Koristi se da inicijalizira statički podatak-član.
- Izvršava se prije main metode u vremenu učitavanja klase.

Primjer statičkog bloka

```
class A2{

 static{System.out.println("statički blok je pozvan");}

 public static void main(String args[]){
  System.out.println("Hello main");
 }
}
```

```
Izlaz: statički blok je pozvan
       Hello main
```

Može li se program izvršiti bez main() metode?

Da, jedan od načina je statički blok ali u prethodnoj verziji JDK ne u JDK 1.7.

```
class A3{
 static{
 System.out.println("statički blok je pozvan");
 System.exit(0);
 }
}
```

```
Izlaz:statički blok je pozvan(ako nije JDK7)
```

U JDK7 i novijim, izlaz će biti:

```
Izlaz:Error: Main method not found in class A3, please define the
main method as:
public static void main(String[] args)
```

7. Ključna riječ this

Postoji više načina upotrebe **ključne riječi this**. U Javi, ovo je **referentna varijabla** koja upućuje na tekući objekt.

Upotreba ključne riječi this
Ovdje je dato 6 upotreba ključne riječi this.

1. this može biti upotrijebljen da uputi na instansnu varijablu tekuće klase.
2. this() može biti upotrijebljen da pozove konstruktor tekuće klase.
3. this može biti upotrijebljen da pozove metod tekuće klase (implicitno).
4. this može biti proslijeđen kao argument u pozivu metode.
5. this može biti proslijeđen kao argument u pozivu konstruktora.
6. this može takođe biti upotrijebljen da vrati instancu tekuće klase.

Napomena: Početnici u Javi ne bi trebalo da koriste više od dve od ovih upotreba ključne riječi this.

7.1 Ključna riječ this može se koristiti da uputi na instansnu varijablu tekuće klase

Ako postoji dvosmislenost između instansne varijable i parametra, ključna riječ this rješava problem dvosmislenosti.
Problem koji se javlja bez ključne riječi this
Sljedeći primjer objašnjava koji se problem javlja ako se ne koristi ključna riječ this:

```
class Student10{
   int id;
   String name;

   student(int id,String name){
   id = id;
   name = name;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student10 s1 = new Student10(111,"Petar");
   Student10 s2 = new Student10(321,"Marko");
   s1.display();
   s2.display();
   }
}
Izlaz: 0 null
       0 null
```

U ovom primjeru, parametar (formalni argumenti) i instansne varijable su iste, zbog čega koristimo ključnu riječ this da razlikujemo lokalnu varijablu i instansnu varijablu.

7.1.1 Rješenje problema pomoću ključne riječi this

```
// primjer za ključnu riječ this
class Student11{
   int id;
   String name;

   Student11(int id,String name){
   this.id = id;
   this.name = name;
   }
   void display(){System.out.println(id+" "+name);}
   public static void main(String args[]){
   Student11 s1 = new Student11(111,"Petar");
   Student11 s2 = new Student11(222,"Marko");
   s1.display();
   s2.display();
}
}
```

```
Izlaz: 111 Petar
       222 Marko
```

Ako su lokalne varijable (formalni argumenti) i instansne varijable različite, nema potrebe da se koristi ključna riječ this, kao u sljedećem programu:

7.1.2 Program gdje ključna riječ this nije potrebna

```
class Student12{
   int id;
   String name;

   Student12(int i,String n){
   id = i;
   name = n;
   }
   void display(){System.out.println(id+" "+name);}
   public static void main(String args[]){
   Student12 e1 = new Student12(111,"Petar");
   Student12 e2 = new Student12(222,"Marko");
   e1.display();
   e2.display();
}
}
```

```
Izlaz: 111 Petar
       222 Marko
```

7.2 this() može biti upotrijebljen da pozove konstruktor tekuće klase

this() poziv konstruktora može se koristiti da pozove konstruktor tekuće klase (ulančavanje konstruktora). Ovaj pristup je bolji ako u klasi ima mnogo konstruktora, a želi se ponovna upotreba tog konstruktora.

```
// Program za this() poziv konstruktora (ulančavanje konstruktora)

class Student13{
   int id;
   String name;
   Student13(){System.out.println("default konstruktor je pozvan");}

   Student13(int id,String name){
   this (); // koristi se da pozove konstruktor tekuće klase
   this.id = id;
   this.name = name;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student13 e1 = new Student13(111,"Petar");
   Student13 e2 = new Student13(222,"Marko");
   e1.display();
   e2.display();
  }
}

Izlaz:
       default konstruktor je pozvan
       default konstruktor je pozvan
       111 Petar
       222 Marko
```

7.2.1 Gdje se koristi this() poziv konstruktora?

this() poziv konstruktora treba koristiti za ponovnu upotrebu (reuse) konstruktora u konstruktoru. On održava vezu između konstruktora tj. koristi se za ulančavanje konstruktora. Pogledajmo primjer koji pokazuje stvarnu upotrebu ključne riječi this.

```
class Student14{
   int id;
   String name;
   String city;

   Student14(int id,String name){
```

```
    this.id = id;
    this.name = name;
    }
    Student14(int id,String name,String city){
    this(id,name); // sada nema potrebe da se inicijalizuje id i name
    this.city=city;
    }
    void display(){System.out.println(id+" "+name+" "+city);}

    public static void main(String args[]){
    Student14 e1 = new Student14(111,"Petar");
    Student14 e2 = new Student14(222,"Marko","Beograd");
    e1.display();
    e2.display();
   }
}
Izlaz: 111 Petar null
       222 Marko Beograd
```

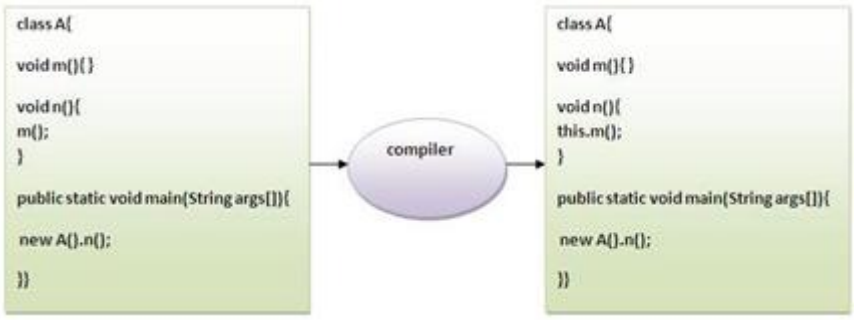
Pravilo: Poziv this() mora biti prva naredba u konstruktoru.

```
class Student15{
    int id;
    String name;
    Student15(){System.out.println("default konstruktor je pozvan");}

    Student15(int id,String name){
    id = id;
    name = name;
    this (); // mora biti prva naredba
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student15 e1 = new Student15(111,"Petar");
    Student15 e2 = new Student15(222,"Marko");
    e1.display();
    e2.display();
   }
}

Izlaz:Compile Time Error
```

7.3 this može biti upotrijebljen da pozove metod tekuće klase (implicitno)

Moguće je pozvati metod tekuće klase pomoću ključne riječi this. Ako se ne upotrijebi ključna riječ this, kompajler će automatski dodati ključnu riječ this dok poziva metod.

Pogledajmo primjer:




```
class S{
  void m(){
  System.out.println("metod je pozvan");
  }
  void n(){
  this.m(); // nije potrebno zato što kompajler to uradi sam.
  }
  void p(){
  n(); // kompajler će dodati this da pozove n() metod kao this.n()
  }
  public static void main(String args[]){
  S s1 = new S();
  s1.p();
  }
}
```

```
Izlaz:method je pozvan
```

7.4 Ključna riječ this može se proslijediti kao argument u metodi

Ključna riječ this može takođe biti proslijeđena kao argument u metodi. Ovo se uglavnom koristi u rukovanju događajima. Pogledajmo primjer:

```
class S2{
  void m(S2 obj){
  System.out.println("metod je pozvan");
  }
  void p(){
```

```
 m(this);
 }

 public static void main(String args[]){
 S2 s1 = new S2();
 s1.p();
 }
}
```

```
Izlaz:metod je pozvan
```

Kada this ne može biti proslijeđen kao argument?
U rukovanju događajima ili u situaciji gdje moramo obezbijediti referencu klase drugoj klasi.

this može biti proslijeđen kao argument u pozivu konstruktora

Ključna riječ this se takođe može proslijediti u konstruktoru. Korisno je ako možemo upotrijebiti jedan objekt u više klasa. Pogledajmo primjer:

```
class B{
 A4 obj;
 B(A4 obj){
  this.obj=obj;
 }
 void display(){
  System.out.println(obj.data); // upotreba podatka-člana klase A4
 }
}

class A4{
 int data=10;
 A4(){
  B b=new B(this);
  b.display();
 }
 public static void main(String args[]){
  A4 a=new A4();
 }
}
```

```
Izlaz:10
```

Ključna riječ this se može upotrijebiti da vrati instancu tekuće klase

Ključnu riječ this možemo vratiti kao naredbu iz metode. U tom slučaju, vraćeni (return) tip metode mora biti tip klase (ne-primitivan). Pogledajmo primjer:

Sintaksa za this koji može biti vraćen kao naredba

```
return_type method_name(){
return this;
}
```

Primjer ključne riječi this koja se vraća kao naredba metode

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}

class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
Izlaz:Hello java
```

Dokazivanje ključne riječi this

Dokažimo da ključna riječ this upućuje na instansnu varijablu tekuće klase. U ovom programu, ispisujemo referentnu varijablu i this, izlaz obe varijable je isti.

```
class A5{
void m(){
System.out.println(this); // ispisuje ID iste reference
}

public static void main(String args[]){
A5 obj=new A5();
System.out.println(obj); // ispisuje ID reference

obj.m();
}
}

Izlaz: A5@22b3ea59
       A5@22b3ea59
```

8. Nasljeđivanje (IS-A) u Javi

Nasljeđivanje u Javi je mehanizam u kojem jedan objekt dobija sve osobine i ponašanja roditeljskog objekta.
Ideja koja stoji iza nasljeđivanja u Javi je ta da se mogu kreirati nove klase koje su izgrađene na postojećim klasama. Kada se nasljeđuje iz postojeće klase, moguće je ponovo upotrijebiti metode i polja roditeljske klase, a takođe je moguće dodati nove metode i polja.
Nasljeđivanje predstavlja IS-A relaciju, takođe poznatu i kao relacija Roditelj-Dijete.

Zbog čega se koristi nasljeđivanje?

- zbog preklapanja (overriding) metoda (takođe i polimorfizma u vremenu izvršavanja)
- zbog ponovne upotrebljivosti (reusability) koda

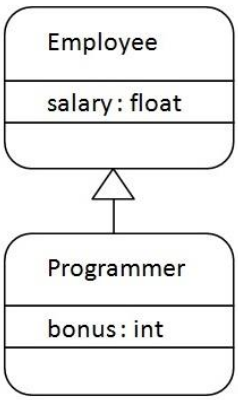
Sintaksa nasljeđivanja

```
class Subclass-name extends Superclass-name
{
  // metode i polja
}
```

Ključna riječ **extends** naznačava da se pravi nova klasa koja se izvodi iz postojeće klase. U terminologiji Jave, klasa koja se nasljeđuje se naziva superklasa. Nova klasa se naziva subklasa.

Objašnjenje jednostavnog primjera nasljeđivanja

Kao što je prikazano na slici, Programmer je subklasa, a Employee je superklasa. Relacija između dve klase je **Programmer IS-A Employee**. To znači da je Programmer tip od Employee.




```
class Employee{
 float salary=40000;
}

class Programmer extends Employee{
```

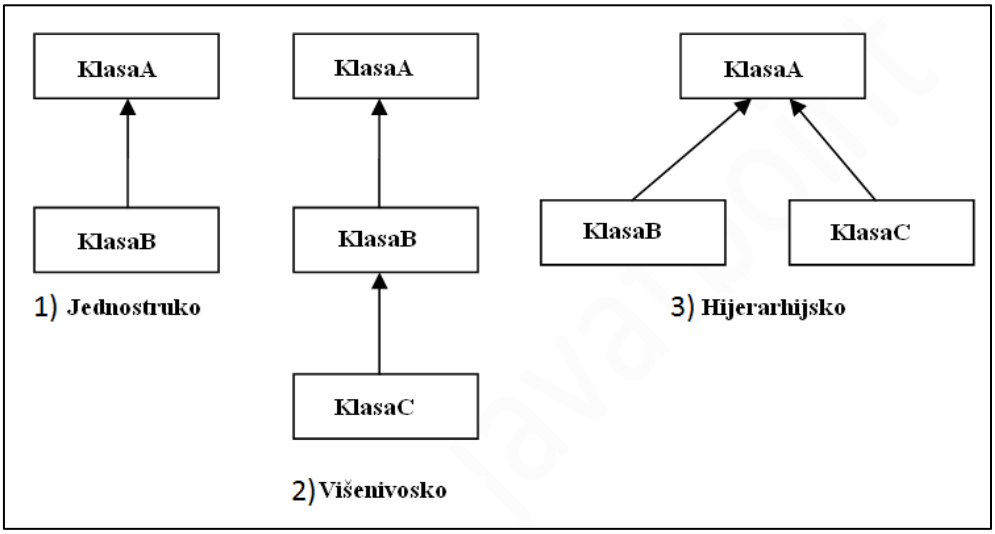
```
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
 Izlaz: Programmer salary is:40000.0
        Bonus of programmer is:10000
```

U ovom primjeru, objekt Programmer može pristupiti polju svoje klase, a također i polju klase Employee tj. imamo ponovnu upotrebljivost koda.

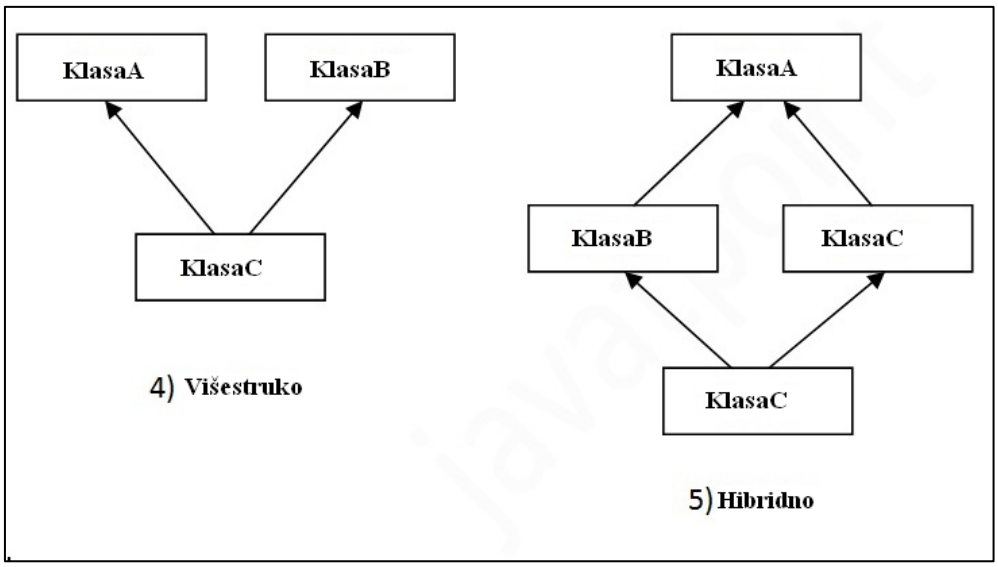
8.1.1 Tipovi nasljeđivanja

Po osnovu klase, u Javi mogu postojati tri tipa nasljeđivanja: jednostruko, višenivosko (multilevel) i hijerarhijsko. U Java programiranju, višestruko i hibridno nasljeđivanje je podržano samo preko interfejsa, o čemu će biti riječi kasnije.



Napomena: Višestruko nasljeđivanje nije podržano u Javi u slučaju klase.

Kada jednu klasu nasljeđuje više klasa to je poznato kao višestruko nasljeđivanje. Napr.:



8.1.1 Zašto višestruko nasljeđivanje nije podržano u Javi?

Da bi se redukovala kompleksnost i pojednostavio jezik, višestruko nasljeđivanje nije podržano u Javi.
Razmotrimo scenario gdje su A, B i C tri klase. Klasa C nasljeđuje klase A i B. Ako klase A i B imaju isti metod i on bude pozvan iz objekta klase-djeteta, postojaće dvosmislenost da li je pozvan metod A ili B klase.
Pošto su greške u vremenu kompajliranja (compile time error) bolje nego greške u vremenu izvršavanja (runtime error), Java će prijaviti compile time error ako naslijedimo 2 klase. Dakle, bez obzira da li je metod isti ili različit, biće prijavljen compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{

 public static void main(String args[]){
   C obj=new C();
   obj.msg(); // Koji će msg() metod biti pozvan?
}
}
```

```
Izlaz: Compile Time Error
```

9. Agregacija (HAS-A) u Javi

Ako klasa ima referencu na neki entitet, to je poznato kao agregacija. Agregacija predstavlja HAS-A relaciju.
Razmotrimo sljedeću situaciju: objekt Employee sadrži mnoge informacije kao što su **id**, **name**, **email** itd. On sadrži još jedan objekt koji se zove **address**, koji sadrži svoje sopstvene informacije kao što su grad, republika ili kanton, država, zipcode itd. kao što je prikazano ovdje:

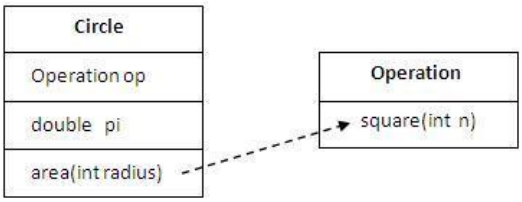
```
class Employee{
int id;
String name;
Address address; //Address je klasa
...
}
```

U ovakvom slučaju, Employee ima referencu na entitet address, tako da je relacija Employee HAS-A address.

Zašto se koristi agregacija?

- zbog ponovne upotrebljivosti (reusability) koda.

Jednostavan primjer agregacije



U ovom primjeru, kreiraćemo referencu klase Operation u klasi Circle.

```
class Operation{
 int square(int n){
  return n*n;
 }
}

class Circle{
 Operation op; //agregacija
 double pi=3.14;

 double area(int radius){
   op=new Operation();
```

```
  int rsquare=op.square(radius); //ponovna upotrebljivost koda
  return pi*rsquare;
 }

 public static void main(String args[]){
  Circle c=new Circle();
  double result=c.area(5);
  System.out.println(result);
 }
}
```

```
Izlaz:78.5
```

Kada se koristi agregacija?

- Ponovna upotreba koda se najbolje postiže pomoću agregacije onda kada nema relacije is-a.
- Nasljeđivanje se treba koristiti samo ako se relacija is-a održava tokom cijelog vremena života uključenog objekta; inače, agregacija je najbolji izbor.

Primjer agregacije

U ovom primjeru, Employee ima objekt od Address, address objekt sadrži svoje sopstvene informacije kao što su grad, republika ili kanton, država itd. U ovom slučaju relacija je Employee HAS-A address.

Address.java

```
public class Address {
String city,state,country;

public Address(String grad, String republika, String drzava) {
   this.grad = grad;
   this.republika = republika;
   this.drzava = drzava;
}
}
```

Emp.java

```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
   this.id = id;
   this.name = name;
   this.address=address;
}
```

```
 void display(){
System.out.println(id+" "+name);
System.out.println(address.grad+" "+address.republika+" "+address.drzava);
}

public static void main(String[] args) {
Address address1=new Address("BL","RS","BiH");
Address address2=new Address("BN","RS","BiH");

Emp e=new Emp(111,"Petar",address1);
Emp e2=new Emp(112,"Marko",address2);

e.display();
e2.display();

}
}
```

```
Izlaz: 111 Petar
       BL RS BiH
       112 Marko
       BN RS BiH
```

10. Preklapanje (overriding) metoda u Javi

Ako podklasa (klasa-dijete) ima isti metod kakav je deklarisan u roditeljskoj klasi, to je poznato kao **preklapanje (overriding) metoda u javi**.
Drugim riječima, ako podklasa obezbjeđuje specifičnu implementaciju metode koju je obezbijedila jedna od roditeljskih klasa, to je poznato kao preklapanje metoda.

Upotreba preklapanja metoda

- Preklapanje metoda se koristi da obezbijedi specifičnu implementaciju metode koja je već obezbijeđena preko svoje superklase.
- Preklapanje metoda se koristi za polimorfizam u vremenu izvršavanja.

Pravila za preklapanje metoda

1. metoda mora imati isto ime kao u roditeljskoj klasi.
2. metoda mora imati isti parametar kao u roditeljskoj klasi.
3. mora biti IS-A relacija (nasljeđivanje).

Problem koji se javlja bez preklapanja metoda

Razmotrimo problem koji se javlja u programu ako ne koristimo preklapanje metode.

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike extends Vehicle{
   public static void main(String args[]){
  Bike obj = new Bike();
  obj.run();
   }
}
```

```
Izlaz: Vehicle is running
```

Problem je u tome da moramo obezbijediti specifičnu implementaciju metode run() u podklasi te stoga koristimo preklapanje metoda.

10.1.1 Primjer 1 preklapanja metoda

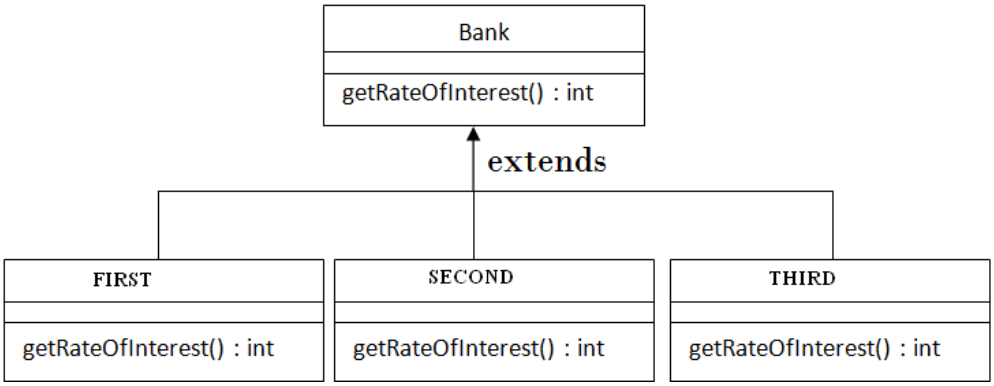
U ovom primjeru, definisaćemo metodu run u podklasi onako kako je definisana u roditeljskoj klasi ali će imati neku specifičnu implementaciju. Ime i parametar metode su isti i postoji IS-A relacija između klasa, tako da postoji preklapanje metoda.

```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}
public static void main(String args[]){
Bike2 obj = new Bike2();
obj.run();
}
```

```
Izlaz:Bike is running safely
```

10.1.2 Primjer 2 preklapanja metoda

Razmotrimo sljedeći scenario: Bank je klasa koja obezbjeđuje funkcionalnost izračunavanja kamatne stope. Ali, kamatna stopa varira u zavisnosti od banke. Na primjer, banke FIRST, SECOND i THIRD mogu obezbijediti 8%, 7% i 9% kamatne stope.




```
class Bank{
int getRateOfInterest(){return 0;}
}

class FIRST extends Bank{
int getRateOfInterest(){return 8;}
}

class SECOND extends Bank{
int getRateOfInterest(){return 7;}
}
class THIRD extends Bank{
int getRateOfInterest(){return 9;}
}

class Test2{
public static void main(String args[]){
FIRST s=new FIRST();
SECOND i=new SECOND();
THIRD a=new THIRD();
System.out.println("FIRST Rate of Interest: "+s.getRateOfInterest());
System.out.println("SECOND Rate of Interest: "+i.getRateOfInterest());
System.out.println("THIRD Rate of Interest: "+a.getRateOfInterest());
}
}
```

```
Izlaz:
FIRST Rate of Interest: 8
SECOND Rate of Interest: 7
THIRD Rate of Interest: 9
```

10.1.3 Može li se preklopiti statička metoda?

Ne, statička metoda se ne može preklopiti. O ovome će biti riječi kasnije, u dijelu koji se odnosi na polimorfizam u vremenu izvršavanja.

Zašto se statička metoda ne može preklopiti?
Zato što je statička metoda vezana za klasu dok je instansna metoda vezana za objekt.

Može li se preklopiti java main metoda?
Ne, zato što je main statička metoda.

10.1.4 Razlika između preopterećenja metode i preklapanja metode u Javi

Postoje tri osnovne razlike između preopterećenja metode i preklapanja metode. To su:

Preopterećenje metode	Preklapanje metode
1) Preopterećenje metode se koristi da poboljša čitljivost programa.	Preklapanje metode se koristi da obezbijedi specifičnu implementaciju metode koju je već obezbijedila njena superklasa.
2) Preopterećenje metode se izvodi unutar klase.	Preklapanje metode se javlja u dve klase koje imaju IS-A relaciju.
3) U slučaju preopterećenja metode parametar mora biti različit.	U slučaju preklapanja metode parametar mora biti isti.

11. Kovarijantni return tip (tip vraćanja)

Kovarijantni return tip specificira da return tip može da se mijenja u istom smjeru kao podklasa.
Prije nego se pojavila Java5, nije bilo moguće preklopiti nijednu metodu promjenom return tipa. Ali sada, poslije Java5, moguće je preklopiti metodu promjenom return tipa ako podklasa preklapa bilo koju metodu čiji je return tip ne-primitivan ali mijenja njen return tip u tip podklase.

Pogledajmo jednostavan primjer:

11.1.1 Jednostavan primjer kovarijantnog return tipa

```
class A{
A get(){return this;}
}

class B1 extends A{
B1 get(){return this;}
void message(){System.out.println("ovo je kovarijantni return tip");}

public static void main(String args[]){
new B1().get().message();
}
}
```

```
Izlaz:ovo je kovarijantni return tip
```

Kao što se može vidjeti u ovom primjeru, return tip metode get() A klase je A ali return tip metode get() B klase je B. Obe metode imaju različit return tip ali to jeste preklapanje metoda. Ovo je poznato kao kovarijantni return tip.

12. Ključna riječ super

Ključna riječ **super** u javi je referentna varijabla koja se koristi da uputi na objekt neposredne roditeljske klase.

Kadgod se kreira instanca podklase, instanca roditeljske klase se kreira implicitno, tj. na nju upućuje referentna varijabla super.

12.1 Upotreba ključne riječi super

1. super se koristi da uputi na instansnu varijablu neposredne roditeljske klase.
2. super() se koristi da pozove konstruktor neposredne roditeljske klase.
3. super se koristi da pozove metodu neposredne roditeljske klase.

12.1.1 super se koristi da uputi na instansnu varijablu neposredne roditeljske klase

Problem bez ključne riječi super

```
class Vehicle{
  int speed=50;
}
class Bike3 extends Vehicle{
  int speed=100;
  void display(){
   System.out.println(speed); //ispisuje brzinu (speed) za Bike
  }
  public static void main(String args[]){
   Bike3 b=new Bike3();
   b.display();
}
}
```

```
Izlaz:100
```

U ovom primjeru obe klase Vehicle i Bike imaju zajedničku osobinu speed. Na instansnu varijablu tekuće klase upućuje instanca po defaultu, ali pošto moramo da uputimo na instansnu varijablu roditeljske klase moramo da koristimo ključnu riječ super da bi se napravila razlika između instansne varijable roditeljske klase i instansne varijable tekuće klase.

Rješenje sa ključnom riječi super

```
class Vehicle{
  int speed=50;
}
 class Bike4 extends Vehicle{
  int speed=100;

  void display(){
   System.out.println(super.speed); //sada će ispisati brzinu (speed) za Vehicle
   }
  public static void main(String args[]){
   Bike4 b=new Bike4();
   b.display();

}
}
```

```
Izlaz:50
```

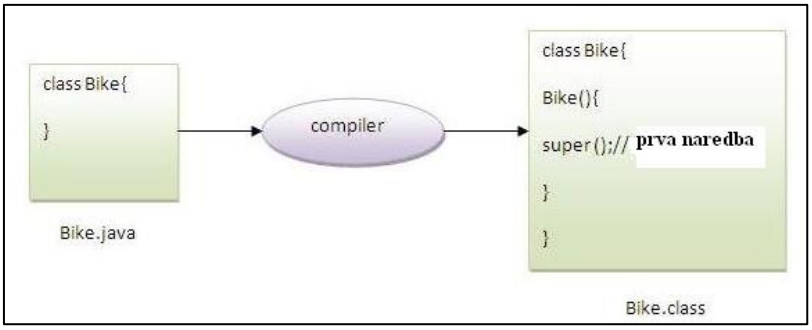
12.1.2 super() se koristi da pozove konstruktor neposredne roditeljske klase.

Ključna riječ super može se također upotrijebiti da pozove konstruktor roditeljske klase kao u sljedećem primjeru:

```
class Vehicle{
  Vehicle(){System.out.println("Vehicle je kreiran");}
}

class Bike5 extends Vehicle{
  Bike5(){
   super(); //poziva konstruktor roditeljske klase
   System.out.println("Bike je kreiran");
   }
  public static void main(String args[]){
   Bike5 b=new Bike5();

}
}
```

```
Izlaz:Vehicle je kreiran
      Bike je kreiran
```

Napomena: super() se dodaje u konstruktor svake klase automatski od strane kompajlera.



Kao što je poznato podrazumijevani (default) konstruktor obezbjeđuje kompajler automatski ali on takođe dodaje super() kao prvu naredbu. Ako kreiramo svoj vlastiti konstruktor, a nemamo ni this() ni super() kao prvu naredbu, kompajler će obezbijediti super() kao prvu naredbu konstruktora.

Primjer za ključnu riječ super gdje je super() implicitno obezbijeđen od strane kompajlera

```
class Vehicle{
  Vehicle(){System.out.println("Vehicle je kreiran");}
}

class Bike6 extends Vehicle{
  int speed;
  Bike6(int speed){
    this.speed=speed;
    System.out.println(speed);
  }
  public static void main(String args[]){
   Bike6 b=new Bike6(10);
 }
}
```

```
Izlaz:Vehicle je kreiran
      10
```

12.1.3 super se koristi da pozove metodu neposredne roditeljske klase

Ključna riječ super može se također upotrijebiti da pozove metodu roditeljske klase. Ovo treba koristiti u slučaju kada podklasa sadrži istu metodu kao roditeljska klasa, kao u sljedećem primjeru:

```
class Person{
void message(){System.out.println("welcome");}
}

class Student16 extends Person{
void message(){System.out.println("welcome to java");}

void display(){
message(); //poziva metodu message() tekuće klase
super.message(); //poziva metodu message() roditeljske klase
}

public static void main(String args[]){
Student16 s=new Student16();
s.display();
}
}
```

```
Izlaz:welcome to java
      welcome
```

U ovom primjeru obe klase Student i Person imaju metodu message(), pa ako pozovemo metodu message() iz klase Student, to će pozvati metodu message() klase Student, a ne klase Person zato što se prioritet daje lokalno.

U slučaju da u podklasi ne postoji metoda kao u roditeljskoj klasi, nema potrebe da se koristi super. U sljedećem primjeru metoda message() je pozvana iz klase Student ali klasa Student nema metodu message(), tako da se metoda message() može pozvati direktno.

12.1.4 Program gdje ključna riječ super nije potrebna

```
class Person{
void message(){System.out.println("welcome");}
}

class Student17 extends Person{

void display(){
message(); //poziva metodu message() roditeljske klase
}

public static void main(String args[]){
Student17 s=new Student17();
s.display();
}
}
```

```
Izlaz:welcome
```

13. Blok inicijalizator instance

Blok inicijalizator instance se koristi da inicijalizira podatak-član instance. On se pokreće svaki put kada se kreira objekt klase.
Inicijalizacija instansne varijable može biti direktna ali moguće je izvršiti i neke dodatne operacije prilikom inicijalizovanja instansne varijable u bloku inicijalizatoru instance.

Zašto se koristi blok inicijalizator instance?

Znamo da je moguće direktno dodijeliti vrijednost podatku-članu instance. Napr.

```
class Bike{
   int speed=100;
}
```

Pretpostavimo da moramo izvršiti neke operacije prilikom dodjeljivanja vrijednosti podatku-članu instance, napr. for petlju koja će popuniti neki kompleksni niz ili rukovanje greškama itd.

Primjer bloka inicijalizatora instance
Pogledajmo jednostavan primjer bloka inicijalizatora instance koji izvršava inicijalizaciju

```
class Bike7{
   int speed;

   Bike7(){System.out.println("brzina je "+speed);}

   {speed=100;}

   public static void main(String args[]){
   Bike7 b1=new Bike7();
   Bike7 b2=new Bike7();
   }
}
```

```
Izlaz:brzina je 100
      brzina je 100
```

Postoje tri mjesta u javi gdje se mogu izvoditi operacije:

1. metoda
2. konstruktor
3. blok

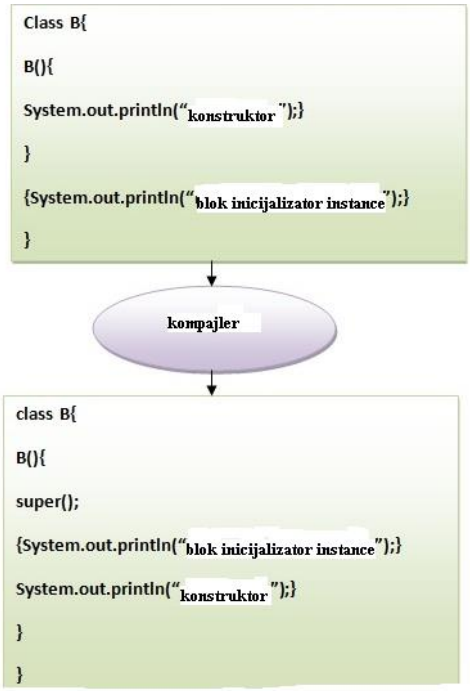
Da li se prvo poziva blok inicijalizator instance ili konstruktor?

```
class Bike8{
   int speed;

   Bike8(){System.out.println("konstruktor je pozvan");}

   {System.out.println("blok inicijalizator instance je pozvan");}

   public static void main(String args[]){
   Bike8 b1=new Bike8();
   Bike8 b2=new Bike8();
   }
}
```

```
Izlaz: blok inicijalizator instance je pozvan
       konstruktor je pozvan
       blok inicijalizator instance je pozvan
       konstruktor je pozvan
```

U ovom primjeru, izgleda kao da je blok inicijalizator instance prvi pozvan, ali zapravo nije. Blok inicijalizator instance je pozvan u vrijeme kreiranja objekta. Java kompajler kopira blok inicijalizator instance u konstruktor nakon prve naredbe super(). Tako je prvo pozvan konstruktor. Ovo je pokazano na sljedećoj slici:

Napomena: java kompajler kopira kod bloka inicijalizatora instance u svaki konstruktor.



Pravila za blok inicijalizator instance:

Postoje uglavnom tri pravila za blok inicijalizator instance. To su:

1. Blok inicijalizator instance je kreiran kada je kreirana instanca klase.
2. Blok inicijalizator instance se poziva nakon što je pozvan konstruktor roditeljske klase (napr. nakon poziva super() konstruktora).
3. Blok inicijalizator instance dolazi po redu po kome se pojavljuje.

13.1.1 Primjer1: blok inicijalizator instance koji se poziva nakon super()

```
class A{
A(){
System.out.println("pozvan je konstruktor roditeljske klase");
}
}
class B2 extends A{
B2(){
super();
System.out.println("pozvan je konstruktor klase-djeteta");
}

{System.out.println("pozvan je blok inicijalizator instance");}

public static void main(String args[]){
B2 b=new B2();
}
}
```

```
Izlaz:pozvan je konstruktor roditeljske klase
      pozvan je blok inicijalizator instance
      pozvan je konstruktor klase-djeteta
```

Primjer2:

```
class A{
A(){
System.out.println("pozvan je konstruktor roditeljske klase");
}
}

class B3 extends A{
B3(){
super();
System.out.println("pozvan je konstruktor klase-djeteta");
}

B3(int a){
super();
System.out.println("pozvan je konstruktor klase-djeteta"+a);
}

{System.out.println("pozvan je blok inicijalizator instance");}

public static void main(String args[]){
B3 b1=new B3();
B3 b2=new B3(10);
}
}
```

```
Izlaz: pozvan je konstruktor roditeljske klase
       pozvan je blok inicijalizator instance
       pozvan je konstruktor klase-djeteta
       pozvan je konstruktor roditeljske klase
       pozvan je blok inicijalizator instance
       pozvan je konstruktor klase-djeteta 10
```

14. Ključna riječ final

Ključna riječ final u javi se koristi da ograniči korisnika. Može se koristiti u mnogim kontekstima. Final može biti:

1. varijabla
2. metoda
3. klasa

Ključna riječ final može se primijeniti sa varijablama, final varijabla koja nema vrijednosti se naziva blank final varijabla ili neinicijalizovana final varijabla. Ona može biti inicijalizovana samo u konstruktoru. Blank final varijabla takođe može biti statička i biće inicijalizovana samo u statičkom bloku. Razmotrimo neke osnovne stvari o ključnoj riječi final.

Napomena: Ključna riječ final

- zaustavlja promjenu vrijednosti
- zaustavlja preklapanje metoda
- zaustavlja nasljeđivanje

14.1.1 Final varijabla

Ako bilo koju varijablu proglasimo za final, ne možemo više mijenjati njenu vrijednost (ona će biti konstanta).

Primjer final varijable

Ovdje imamo final varijablu speedlimit, pokušaćemo promijeniti njenu vrijednost, ali to neće biti moguće zato što jednom dodijeljena vrijednost final varijabli više ne može biti promijenjena.

```
class Bike9{
 final int speedlimit=90; //final varijabla
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
} //kraj klase
```

```
Izlaz:Compile Time Error
```

14.1.2 Final metoda

Ako se bilo koja metoda proglasi za final, ona se ne može preklopiti.

Primjer final metode

```
class Bike{
  final void run(){System.out.println("vozi");}
}
class Honda extends Bike{
   void run(){System.out.println("vozi sigurno 100km/h");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

```
Izlaz:Compile Time Error
```

14.1.3 Final klasa

Ako bilo koju klasu proglasimo za final, ona se ne može naslijediti.

Primjer final klase

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("vozi sigurno 100km/h");}

  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

```
Izlaz:Compile Time Error
```

Da li se final metoda može naslijediti?

Da, final metoda se može naslijediti ali se ne može preklopiti. Napr.:

```
class Bike{
  final void run(){System.out.println("vozi...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

```
Izlaz:vozi...
```

14.2 Šta je blank ili neinicijalizovana final varijabla?

Final varijabla koja nije inicijalizovana u vrijeme deklarisanja je poznata kao blank final varijabla.
Ako želimo da kreiramo varijablu koja se inicijalizuje u vrijeme kreiranja objekta i pošto je jednom inicijalizovana više se ne može mijenjati, ovo je koristan način. Napr. PAN CARD broj zaposlenika.
Može se inicijalizovati samo u konstruktoru.

14.2.1 Primjer blank final varijable

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

Može li se inicijalizovati blank final varijabla?

Da, ali samo u konstruktoru. Napr.:

```
class Bike10{
  final int speedlimit; //blank final varijabla

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }
}
```

```
Izlaz:70
```

14.2.1 Statička blank final varijabla

Statička final varijabla koja nije inicijalizovana u vrijeme deklarisanja je poznata kao statička blank final varijabla. Ona se može inicijalizovati samo u statičkom bloku.

Primjer statičke blank final varijable

```
class A{
  static final int data; //statička blank final varijabla
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
 }
}
```

14.2.1 Šta je final parametar?

Ako deklarišemo bilo koji parametar kao final, ne možemo mijenjati njegovu vrijednost.

```
class Bike11{
  int cube(final int n){
   n=n+2; //ne može se mijenjati pošto je n final
   n*n*n;
  }
  public static void main(String args[]){
    Bike11 b=new Bike11();
    b.cube(5);
 }
}
Izlaz:Compile Time Error
```

Da li se konstruktor može deklarisati kao fnal?
Ne, pošto se konstruktor nikad ne nasljeđuje.

15. Polimorfizam

Polimorfizam u Javi je koncept pomoću kojeg možemo izvesti *jednu akciju na različite načine*. Riječ polimorfizam se izvodi iz dve grčke riječi: poly i morphe. Riječ "poly" znači više, a "morphe" znači oblik. Dakle, polimorfizam znači više oblika.

Postoje dva tipa polimorfizma u Javi: polimorfizam u vremenu kompajliranja i polimorfizam u vremenu izvršavanja. Polimorfizam u Javi se može izvoditi pomoću preopterećenja metode i preklapanja metode.
Ako se preopteretі statička metoda u Javi, to je primjer polimorfizma u vremenu kompajliranja. Ovdje ćemo se fokusirati na polimorfizam u vremenu izvršavanja.

15.1.1 Polimorfizam u vremenu izvršavanja

Polimorfizam u vremenu izvršavanja (runtime polymorphism) ili **dinamičko otpremanje metode (Dynamic Method Dispatch)** je proces u kojem se poziv preklopljene metode rješava u vremenu izvršavanja, a ne u vremenu kompajliranja.
U ovom procesu, preklopljena metoda se poziva preko referentne varijable superklase. Određivanje koja će metoda biti pozvana se zasniva na objektu na koji ukazuje referentna varijabla.

Prije runtime polimorfizma treba shvatiti pojam podizanja (upcasting).

15.2 Upcasting

Kada referentna varijabla roditeljske klase upućuje na objekt klase-djeteta, to je poznato kao upcasting.

Na primjer:

```
class A{}
class B extends A{}

A a=new B(); //upcasting
```

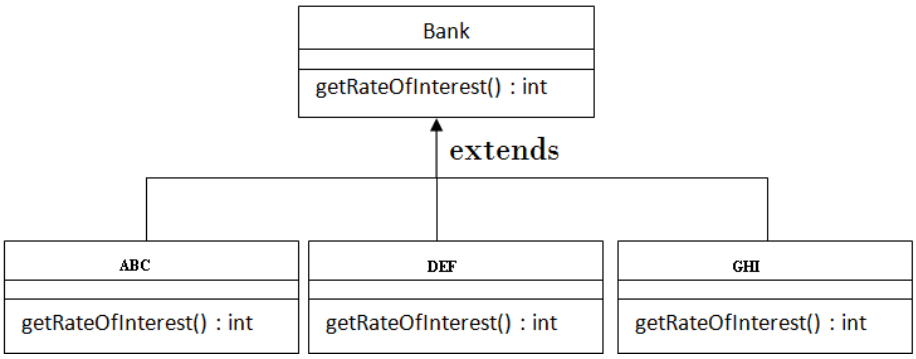
15.2.1 Primjer1 Java runtime polimorfizma

U ovom primjeru, kreiraćemo dve klase Bike i Splender. Splender klasa nasljeđuje Bike klasu i preklapa njenu run() metodu. Metodu run pozivamo pomoću referentne varijable roditeljske klase. Pošto ona upućuje na objekt podklase, a metoda podklase preklapa metodu roditeljske klase, metoda podklase se poziva u vremenu izvršavanja.
Pošto pozivanje metode određuje JVM a ne kompajler, to je poznato kao runtime polimorfizam.

```
class Bike{
  void run(){System.out.println("vozi");}
}
class Splender extends Bike{
  void run(){System.out.println("vozi sigurno 60km/h");}

  public static void main(String args[]){
    Bike b = new Splender(); //upcasting
    b.run();
  }
}
Izlaz:vozi sigurno 60km/h.
```

15.2.2 Primjer2 Java runtime polimorfizma

Razmotrimo sljedeći scenario: Bank je klasa koja obezbjeđuje metodu za dobijanje kamatne stope. Ali, kamatna stopa se može razlikovati u zavisnosti od banke. Na primjer, banke ABC, DEF i GHI mogu davati kamatnu stopu od 8%, 7% i 9%.



Napomena: Ovo je također dato u preklapanju metoda ali tamo nije bilo upcastinga.

```
class Bank{
int getRateOfInterest(){return 0;}
}

class ABC extends Bank{
int getRateOfInterest(){return 8;}
}

class DEF extends Bank{
int getRateOfInterest(){return 7;}
}
class GHI extends Bank{
int getRateOfInterest(){return 9;}
}

class Test3{
public static void main(String args[]){
Bank b1=new ABC();
Bank b2=new DEF();
Bank b3=new GHI();
System.out.println("ABC kamatna stopa: "+b1.getRateOfInterest());
System.out.println("DEF kamatna stopa: "+b2.getRateOfInterest());
System.out.println("GHI kamatna stopa: "+b3.getRateOfInterest());
}
}
Izlaz:
ABC kamatna stopa: 8
DEF kamatna stopa: 7
GHI kamatna stopa: 9
```

15.3 Java runtime polimorfizam sa podatkom-članom

Preklopljena je metoda, a ne podaci-članovi, tako da runtime polimorfizam ne može biti postignut pomoću podataka-članova.
U sljedećem primjeru, obe klase imaju podatak-član speedlimit, podatku-članu pristupamo pomoću referentne varijable roditeljske klase koja upućuje na objekt podklase. Pošto pristupamo podatku-članu koji nije preklopljen, to će značiti da će se uvijek pristupiti podatku-članu roditeljske klase.

Pravilo: Runtime polimorfizam se ne može postići pomoću podataka-članova.

```
class Bike{
 int speedlimit=90;
}
class Honda3 extends Bike{
 int speedlimit=150;

 public static void main(String args[]){
  Bike obj=new Honda3();
  System.out.println(obj.speedlimit); //90
}
Izlaz:90
```

15.4 Java runtime polimorfizam sa višenivoskim (multilevel) nasljeđivanjem

Pogledajmo jednostavan primjer runtime polimorfizma sa multilevel nasljeđivanjem.

```
class Animal{
void eat(){System.out.println("eating");}
}

class Dog extends Animal{
void eat(){System.out.println("eating fruits");}
}

class BabyDog extends Dog{
void eat(){System.out.println("drinking milk");}

public static void main(String args[]){
Animal a1,a2,a3;
a1=new Animal();
a2=new Dog();
a3=new BabyDog();
```

```
a1.eat();
a2.eat();
a3.eat();
}
}
Izlaz: eating
       eating fruits
       drinking milk
```

15.4.1 Primjer1

```
class Animal{
void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
void eat(){System.out.println("dog is eating...");}
}

class BabyDog1 extends Dog{
public static void main(String args[]){
Animal a=new BabyDog1();
a.eat();
}}
Izlaz: dog is eating...
```

Pošto BabyDog ne preklapa metodu eat(), to znači da je pozvana metoda eat() klase Dog.

16. Statičko i dinamičko povezivanje

Veza poziva metode sa tijelom metode se naziva povezivanje.
Postoje dva tipa povezivanja

1. statičko povezivanje (poznato i kao rano povezivanje).
2. dinamičko povezivanje (poznato i kao kasno povezivanje).

Pojam tipa
Podsjetimo se pojma tipa instance

1) Varijable imaju tip
Svaka varijabla ima tip, koji može biti primitivan ili ne-primitivan.

```
int data=30;
```

Ovdje je varijabla data tipa int.

2) Reference imaju tip

```
class Dog{
 public static void main(String args[]){
  Dog d1; //Ovdje je d1 tipa Dog
 }
}
```

3) Objekti imaju tip
Objekt je instanca određene java klase, ali je također instanca svoje superklase.

```
class Animal{}

class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }
}
```

Ovdje je d1 instanca klase Dog, ali je također instanca klase Animal.

16.1 Statičko povezivanje

Kada je tip objekta određen u vremenu kompajliranja (od strane kompajlera), to je poznato kao statičko povezivanje.
Ako u klasi postoji bilo koja private, final ili static metoda, postoji i statičko povezivanje.

Primjer statičkog povezivanja

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}
 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

16.2 Dinamičko povezivanje

Kada je tip objekta određen u vremenu izvršavanja (runtime), to je poznato kao dinamičko povezivanje.

16.2.1 Primjer dinamičkog povezivanja

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

```
Izlaz:dog is eating...
```

U ovom primjeru tip objekta ne može odrediti kompajler, zato što je instanca klase Dog takođe instanca klase Animal. Tako kompajler ne zna njen tip, već samo njen osnovni tip.

17. Operator instanceof

Java instanceof operator se koristi da testira da li je objekt instanca specificiranog tipa (klasa ili podklasa ili interfejs).

Instanceof u javi je takođe poznat kao *operator poređenja po tipu* jer upoređuje instancu sa tipom. On vraća vrijednost true ili false. Ako primijenimo instanceof operator na bilo koju varijablu koja ima vrijednost null, on će vratiti false.

17.1.1 Primjer1 za java instanceof operator

Pogledajmo jednostavan primjer ovog operatora gdje on testira tekuću klasu.

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple); //true
 }
}
```

```
Izlaz:true
```

Objekt tipa podklase je takođe tipa roditeljske klase. Na primjer, ako Dog nasljeđuje (extends) Animal tada objekt od Dog može da upućuje ili na Dog ili na Animal klasu.

17.1.2 Primjer2 za java instanceof operator

```
class Animal{}
class Dog1 extends Animal{ //Dog nasljeđuje Animal

 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal); //true
 }
}
```

```
Izlaz:true
```

17.1.3 Operator instanceof sa varijablom koja ima vrijednost null

Ako primijenimo instanceof operator sa varijablom koja ima vrijednost null, on će vratiti false.
Pogledajmo takav primjer.

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2); //false
 }
}
```

```
Izlaz:false
```

17.1.4 Downcasting sa java instanceof operatorom

Kada tip podklase upućuje na objekt roditeljske klase, to je poznato kao downcasting. Ako se to izvede direktno, kompajler će dati compile time error. Ako se to izvede pomoću typecastinga, biće izbačen ClassCastException u runtime-u. Ali ako koristimo instanceof operator, downcasting je moguć.

Dog d=new Animal(); //compile time error

Ako izvedemo downcasting pomoću typecastinga, ClassCastException će biti izbačen u runtime.
Dog d=(Dog)new Animal(); //uspješno kompajlirano ali ClassCastException je izbačen u runtime-u

Napomena: Typecasting u Javi je dodjeljivanje jednog tipa, klase ili interfejsa, u drugi tip tj. drugu klasu ili interfejs. Napr. neka je Base superklasa, a Derived podklasa.

Pogledajmo ovakav kod:

```
Base b = new Derived(); //referentna varijabla klase Base kreira
objekt klase Derived
Derived d = b; //compile time error, potreban je typecasting
Derived d = (Derived) b; // typecasting Base u Derived
```

17.1.5 Mogućnost downcastinga sa instanceof

Pogledajmo primjer, gdje je downcasting moguć pomoću instanceof operatora.

```
class Animal { }

class Dog3 extends Animal {
  static void method(Animal a) {
    if(a instanceof Dog3){
      Dog3 d=(Dog3)a; //downcasting
      System.out.println("ok downcasting izveden");
    }
  }

  public static void main (String [] args) {
    Animal a=new Dog3();
    Dog3.method(a);
  }

 }
```

```
Izlaz:ok downcasting izveden
```

17.1.6 Downcasting bez upotrebe instanceof

Downcasting se također može izvesti bez upotrebe instanceof operatora kao u sljedećem primjeru:

```
class Animal { }
class Dog4 extends Animal {
  static void method(Animal a) {
      Dog4 d=(Dog4)a; //downcasting
      System.out.println("ok downcasting izveden");
  }
   public static void main (String [] args) {
    Animal a=new Dog4();
    Dog4.method(a);
  }
}
```

```
Izlaz:ok downcasting izveden
```

Razmotrimo ovo detaljnije, stvarni objekt koji je označen kao a, je objekt klase Dog. Ako ga downcastujemo, to je u redu. Ali šta će se desiti ako napišemo:

```
Animal a=new Animal();
Dog.method(a); //sada imamo ClassCastException ali ne u slučaju instanceof operatora
```

17.2 Razumijevanje stvarne upotrebe operatora instanceof

Pogledajmo stvarnu upotrebu ključne riječi instanceof u sljedećem primjeru.

```
interface Printable{}
class A implements Printable{
public void a(){System.out.println("a metoda");}
}
class B implements Printable{
public void b(){System.out.println("b metoda");}
}

class Call{
void invoke(Printable p){ //upcasting
if(p instanceof A){
A a=(A)p; //Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p; //Downcasting
b.b();
}

}
} //kraj klase Call

class Test4{
public static void main(String args[]){
Printable p=new B();
Call c=new Call();
c.invoke(p);
}
}
```

```
Izlaz: b metoda
```

18. Apstraktna klasa u Javi

Klasa koja je deklarisana sa ključnom riječi abstract, je poznata kao apstraktna klasa u javi. Ona može imati apstraktne i ne-apstraktne metode (metode sa tijelom).
Prije upoznavanja sa java apstraktnom klasom, prvo treba shvatiti pojam apstrakcije u javi.

Apstrakcija u javi

Apstrakcija je proces skrivanja detalja implementacije i pokazivanja korisniku samo funkcionalnosti.
Drugim riječima, korisniku se pokazuju samo važne stvari, a unutrašnji detalji ostaju skriveni. Napr. pri slanju sms-a, mi samo kucamo tekst i šaljemo poruku. Nije nam poznato interno procesiranje vezano za slanje poruke.
Apstrakcija nam dopušta da se fokusiramo na to što objekt radi umjesto na kako to radi.

Načini za postizanje apstrakcije

Postoje dva načina za postizanje apstrakcije u javi

1. Apstraktna klasa (0 do 100%)
2. Interfejs (100%)

18.1 Apstraktna klasa u javi

Klasa koja je deklarisana kao apstraktna naziva se **apstraktna klasa**. Ona mora biti naslijeđena i njen metod implementiran. Ona ne može biti instancirana.

Apstraktna klasa je neka vrsta fantomske klase. Ona može prosljeđivati metode i varijable, ali sama nikada ne može biti instancirana, tj. nije moguće kreirati objekt apstraktne klase. U tom smislu, apstraktna klasa je poput interfejsa, ali za razliku od njega ona može sadržavati metode koji nisu apstraktni. Takođe može sadržavati deklaracije podataka koji nisu konstante.

Svaka klasa koja sadrži jedan ili više apstraktnih metoda mora biti deklarisana kao apstraktna. U apstraktnim klasama (za razliku od interfejsa) modifikator abstract mora biti primijenjen na svaki apstraktni metod.

Apstraktne klase se ponašaju kao „držači mjesta“ (placeholderi) u hijerarhiji klasa. Na primjer, apstraktna klasa može sadržavati djelimičnu deskripciju koju nasljeđuju svi njeni potomci u hijerarhiji klasa. Njena djeca, koja su mnogo određenija, popunjavaju praznine.
Apstraktni metod nikada ne može biti static.

18.1.1 Primjer apstraktne klase

```
abstract class A{}
```

Apstraktni metod

Metod koji je deklarisan kao apstraktan i koji nema implementaciju se naziva apstraktni metod.

Primjer apstraktnog metoda

```
abstract void printStatus();  //nema tijela i ima abstract
```

Primjer apstraktne klase koja ima apstraktni metod

U ovom primjeru, apstraktna klasa Bike sadrži samo jedan apstraktni metod run. Njegovu implementaciju obezbjeđuje klasa Honda.

```
abstract class Bike{
  abstract void run();
}

class Honda4 extends Bike{
void run(){System.out.println("running safely..");}

public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Izlaz: running safely..

18.1.2 Razumijevanje stvarnog scenarija apstraktne klase

U ovom primjeru, Shape je apstraktna klasa, a njenu implementaciju obezbjeđuju klase Rectangle i Circle. Najčešće, mi ne znamo ništa o implementaciji klase (tj. ona je skrivena za krajnjeg korisnika) a objekt implementacione klase obezbjeđuje **factory metod**.

Factory metod je metod koji vraća instancu klase. O factory metodu ćemo govoriti kasnije. U ovom primjeru, ako kreiramo instancu klase Rectangle, biće pozvan draw() metod klase Rectangle.

```
abstract class Shape{
abstract void draw();
}
//U stvarnom scenariju, implementaciju obezbjeđuje neko drugi tj. ona je nepoznata
krajnjem korisniku
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
```

```
}

class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}

//U stvarnom scenariju, metod poziva programer ili korisnik
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();  //U stvarnom scenariju, objekt obezbjeđuje metod napr.
getShape() metod
s.draw();
}
}
```

Izlaz: `drawing circle`

18.1.2 Drugi primjer apstraktne klase u javi

```
abstract class Bank{
abstract int getRateOfInterest();
}

class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 7;}
}

class TestBank{
public static void main(String args[]){
Bank b=new SBI();  //ako je objekt PNB, biće pozvan metod od PNB
int interest=b.getRateOfInterest();
System.out.println("Rate of Interest is: "+interest+" %");
}}
```

Izlaz: `Rate of Interest is: 7 %`

18.2 Apstraktna klasa koja ima konstruktor, podatak-član, metode itd.

Apstraktna klasa može imati podatak-član, apstraktni metod, tijelo metoda, konstruktor i čak main() metod.

```
//primjer apstraktne klase koja ima tijelo metoda
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }

 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

```
Izlaz:  bike is created
        running safely..
        gear changed
```

18.3 Pravilo: Ako postoji bar jedan apstraktni metod u klasi, ta klasa mora biti apstraktna.

```
class Bike12{
abstract void run();
}
```

Izlaz: compile time error

Pravilo: Ako nasljeđujemo bilo koju apstraktnu klasu koja ima apstraktni metod, moramo obezbijediti implementaciju tog metoda ili učiniti tu klasu apstraktnom.

18.3.1 Još jedan stvarni scenario za apstraktnu klasu

Apstraktna klasa može također biti upotrijebljena da obezbijedi neke implementacije interfejsa. U tom slučaju, krajnji korisnik ne mora biti prinuđen da preklopi sve metode interfejsa.

Napomena: Početnik u javi , neka prvo pročita poglavlje o interfejsu.

```
interface A{
void a();
void b();
void c();
void d();
}

abstract class B implements A{
public void c(){System.out.println("I am C");}
}

class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}
}
```

```
Izlaz: I am a
       I am b
       I am c
       I am d
```

19. Interfejs u Javi

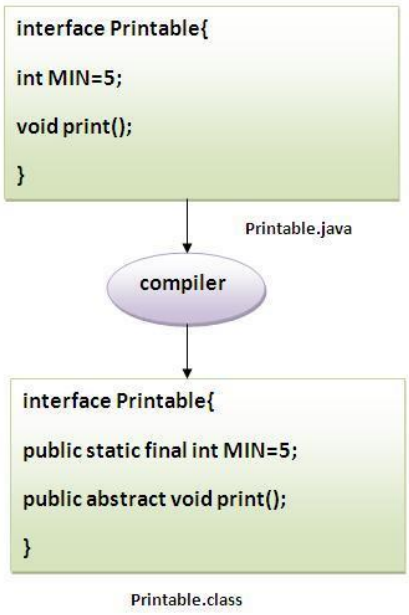
Interfejs u javi je nacrt (blueprint) klase. On ima samo statičke konstante i apstraktne metode.
Interfejs u javi je **mehanizam za postizanje potpune apstrakcije**. U java interfejsu mogu postojati samo apstraktni metodi bez tijela metoda. On se koristi za postizanje pune apstrakcije i višestrukog nasljeđivanja u Javi.
Java Interfejs također **predstavlja IS-A relaciju**.
On ne može biti instanciran kao ni apstraktna klasa.

19.1 Zašto se koristi interfejs?

Postoje uglavnom tri razloga za upotrebu interfejsa. To su:

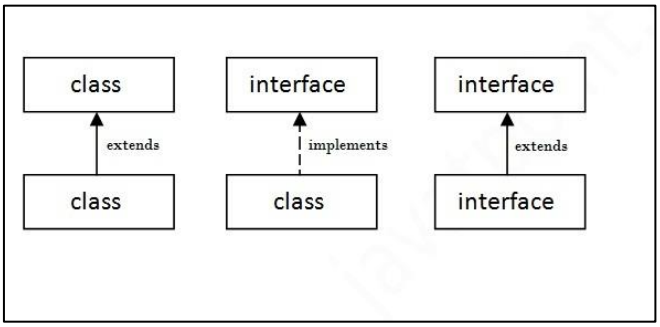
- Koristi se za postizanje pune apstrakcije.
- Pomoću interfejsa, moguće je podržati funkcionalnost višestrukog nasljeđivanja.
- Može se koristiti za postizanje tzv. labavog spajanja (loose coupling).

Napomena: Java kompajler dodaje ključne riječi public i abstract prije metoda interfejsa i ključne riječi public, static i final prije podataka-članova.
Drugim riječima, polja interfejsa su public, static i final po default-u, a metodi su public i abstract.



19.2 Razumijevanje odnosa između klasa i interfejsa

Kao što je prikazano na sljedećoj slici, klasa nasljeđuje drugu klasu, interfejs nasljeđuje drugi interfejs ali k**lasa implementira interfejs**.



19.2.1 Jednostavan primjer java interfejsa

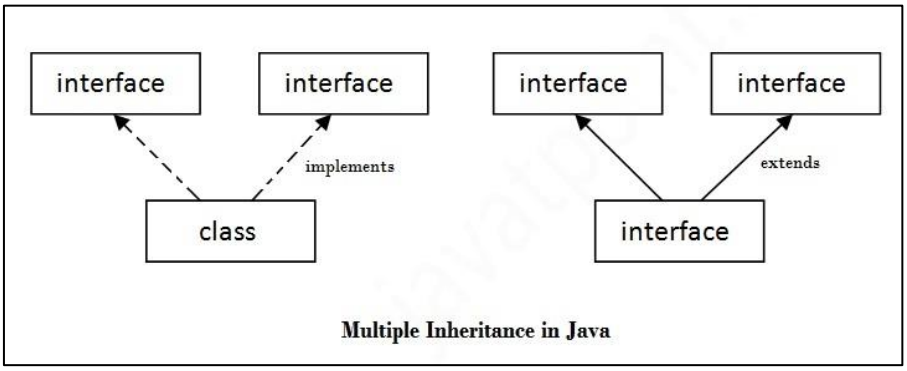
U ovom primjeru, interfejs Printable ima samo jedan metod, njegova implementacija je obezbjeđena u klasi A.

```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Izlaz: Hello

19.3 Višestruko nasljeđivanje u Javi pomoću interfejsa

Ako klasa implementira više interfejsa, ili interfejs nasljeđuje više interfejsa to je poznato kao višestruko nasljeđivanje (multiple inheritance).




```
interface Printable{
void print();
}

interface Showable{
void show();
}

class A7 implements Printable,Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Izlaz: Hello
Welcome

Zašto višestruko nasljeđivanje nije podržano preko klasa u javi ali je moguće pomoću interfejsa?

Kao što je rečeno u poglavlju o nasljeđivanju, višestruko nasljeđivanje nije podržano u slučaju klasa. Međutim, podržano je u slučaju interfejsa zato što nema dvosmislenosti jer je implementacija obezbijeđena preko implementacione klase.
Na primjer:

```
interface Printable{
void print();
}

interface Showable{
void print();
}

class testinterface1 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){
testinterface1 obj = new testinterface1();
obj.print();
 }
}
```

Izlaz: Hello

Kao što se može vidjeti u ovom primjeru, Printable i Showable interfejs imaju iste metode ali njihovu implementaciju obezbjeđuje klasa A, tako da nema dvosmislenosti.

19.4 Nasljeđivanje interfejsa

Klasa implementira interfejs ali jedan interfejs nasljeđuje drugi interfejs.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class Testinterface2 implements Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
 }
}
```

Izlaz: Hello
Welcome

19.4.1 Šta je marker ili tagovani interfejs?

Interfejs koji nema članova je poznat kao marker ili tagovani interfejs. Na primjer: Serializable, Cloneable, Remote itd. Oni se koriste da obezbijede neke suštinske informacije za JVM tako da JVM može izvoditi neke korisne operacije.

```
// Kako je napisan interfejs Serializable?
public interface Serializable{
}
```

19.4.2 Ugniježđeni (nested) interfejs u javi

Napomena: Jedan interfejs može imati drugi interfejs i to je poznato kao ugniježđeni interfejs.
Na primjer:

```
interface printable{
 void print();
 interface MessagePrintable{
   void msg();
 }
}
```

20. Razlika između apstraktne klase i interfejsa

Apstraktna klasa i interfejs se koriste za postizanje apstrakcije gdje možemo deklarisati apstraktne metode.

Apstraktna klasa i interfejs se ne mogu instancirati.
Ali postoje mnoge razlike između apstraktne klase i interfejsa koje su date u sljedećoj tabeli.

Apstraktna klasa	Interfejs
1) Apstraktna klasa može **imati apstraktne i ne-apstraktne** metode.	Interfejs može imati **samo apstraktne** metode.
2) Apstraktna klasa **ne podržava višestruko nasljeđivanje**.	Interfejs **podržava višestruko nasljeđivanje**.
3) Apstraktna klasa **može imati final, non-final, static i non-static varijable**.	Interfejs ima **samo static i final varijable**.
4) Apstraktna klasa **može imati static metode, main metod i konstruktor**.	Interfejs **ne može imati static metode, main metod ili konstruktor**.
5) Apstraktna klasa **može obezbijediti implementaciju interfejsa**.	Interfejs **ne može obezbijediti implementaciju apstraktne klase**.
6) **Ključna riječ abstract** se koristi da deklariše apstraktnu klasu.	**Ključna riječ interface** se koristi da deklariše interfejs.

7) **Primjer:**
public class Shape{
public abstract void draw();
}
	Primjer:
	public interface Drawable{
	void draw();
	}

Jednostavno, apstraktna klasa postiže djelimičnu apstrakciju (0 do 100%) dok interfejs postiže punu apstrakciju (100%).

20.1 Primjer apstraktne klase i interfejsa u javi

Pogledajmo jednostavan primjer gdje koristimo i interfejs i apstraktnu klasu.

```
//Kreiranje interfejsa koji ima 4 metoda
interface A{
void a(); //po default-u, public i abstract
void b();
void c();
void d();
}


//Kreiranje apstraktne klase koja obezbjeđuje implementaciju jednog metoda A interfejsa
abstract class B implements A{
public void c(){System.out.println("I am C");}
}


//Kreiranje podklase apstraktne klase, sada trebamo da obezbijedimo implementaciju ostalih //metoda
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}


//Kreiranje test klase koja poziva metode A interfejsa
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Izlaz: I am a
I am b
I am c
I am d

21. Java package (paket)

Java package (paket) je grupa sličnih tipova klasa, interfejsa i pod-paketa.

Paket u javi može biti kategorisan u dve forme, ugrađeni paket i korisnički definisani paket.

Postoje mnogi ugrađeni paketi kao što su java, lang, awt, javax, swing, net, io, util, sql itd.

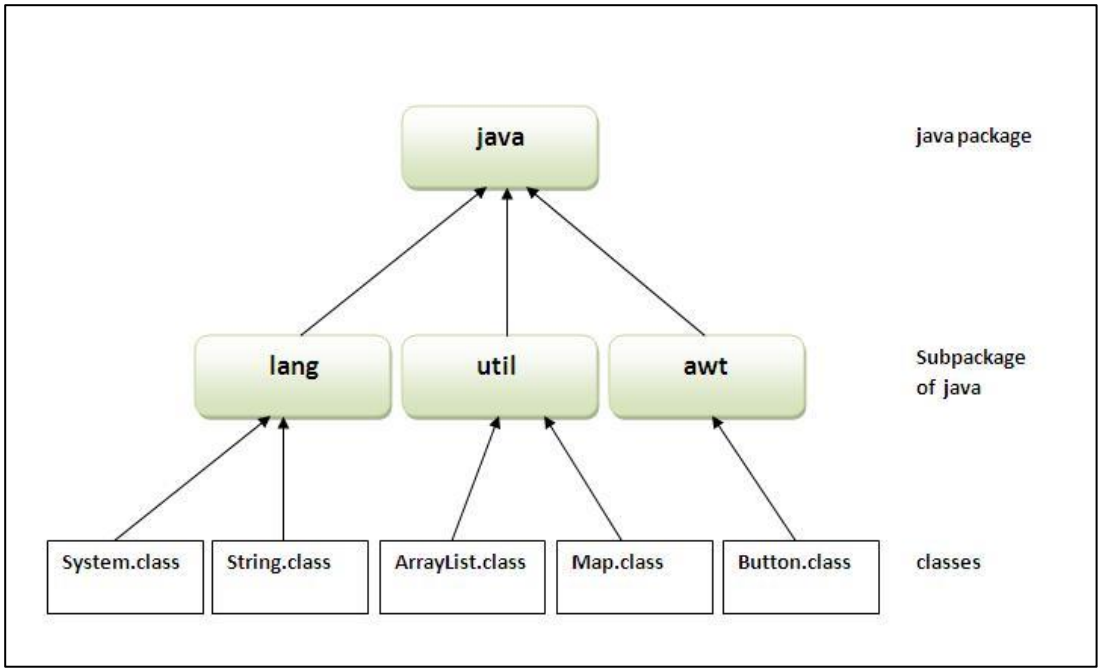
Ovdje, detaljno ćemo razmotriti kreiranje i upotrebu korisnički definisanih paketa.

21.1.1 Prednosti java paketa

1) Java paket se koristi da kategoriše klase i interfejse tako da se oni mogu lakše održavati.

2) Java paket obezbjeđuje zaštitu pristupa.

3) Java paket otklanja kolizije u imenima.



21.2 Jednostavan primjer java paketa

Ključna riječ **package** se koristi za kreiranje paketa u javi.

```
//sačuvano kao Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

21.3 Kako pristupiti paketu iz drugog paketa?

Postoje tri načina za pristup paketu izvan paketa.

1. import package.*;
2. import package.classname;
3. puno kvalifikovano ime.

21.3.1 Pomoću packagename.*

Ako koristimo package.* tada će sve klase i interfejsi ovog paketa biti dostupni ali ne i podpaketi.
Ključna riječ import se koristi da učini klase i interfejse drugog paketa dostupnim tekućem paketu.

Primjer paketa koji importuje packagename.*

```
//sačuvano od A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//sačuvano od B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
```

```
  A obj = new A();
  obj.msg();
 }
}
```

Izlaz: Hello

21.3.2 Pomoću packagename.classname

Ako importujemo package.classname tada će samo deklarisana klasa ovog paketa biti dostupna.

Primjer paketa pomoću import package.classname

```
//sačuvano od A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//sačuvano od B.java

package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Izlaz: Hello

21.3.3 Pomoću punog kvalifikovanog imena

Ako koristimo puno kvalifikovano ime tada će samo deklarisana klasa ovog paketa biti dostupna. Ovdje nema potrebe za importom. Ali zato je potrebno koristiti puno kvalifikovano ime svaki put kada se pristupa klasi ili interfejsu.
Ovo se obično koristi kada dva paketa imaju isto ime klase napr. java.util i java.sql paket sadrže Date klasu.

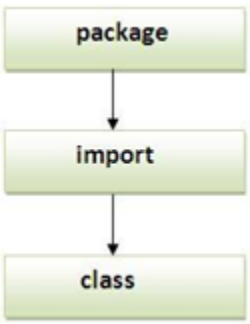
21.3.4 Primjer paketa gdje se koristi puno kvalifikovano ime

```
//sačuvano od A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//sačuvano od B.java

package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A(); //korištenje punog kvalifikovanog imena
   obj.msg();
  }
}
```

Izlaz: Hello

Napomena: Ako importujemo paket, podpaketi neće biti importovani.
Ako importujemo paket, sve klase i interfejsi tog paketa će biti importovani izuzimajući klase i interfejse podpaketa. Dakle, potrebno je takođe importovati i podpakete.

Napomena: Sekvenca programa mora biti paket zatim import zatim klasa.



21.4 Podpaket (subpackage) u javi

Paket unutar paketa se naziva **podpaket (subpackage)**. On se kreira kada treba **dalje kategorizovati paket**.

Pogledajmo primjer, Sun Microsystem je definisao paket pod imenom java koji sadrži mnoge klase kao što su System, String, Reader, Writer, Socket itd. Ove klase predstavljaju određenu grupu napr. Reader i Writer klase služe za ulazno/izlazne operacije, Socket i ServerSocket klase su za umrežavanje itd. Tako, Sun je podkategorizovao java paket u podpakete kao što su lang, net, io itd. i stavio klase koje se odnose na ulaz/izlaz u io paket, Server i ServerSocket klase u net pakete itd.

Standard u definisanju paketa je domain.company.package

21.4.1 Primjer podpaketa

```
package com.company.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

Izlaz: Hello subpackage

Pravilo: Može biti samo jedna public klasa u java source fajlu i mora biti sačuvana po public imenu klase.

```
//sačuvano kao C.java inače Compile Time Error
class A{}
class B{}
public class C{}
```

21.5 Kako staviti dve public klase u jedan paket?

Ako želimo staviti dve public klase u jedan paket, moramo imati dva java source fajla koja sadrže jednu public klasu, ali zadržati isto ime paketa. Na primjer:

```
//sačuvaj kao A.java
   package company;
   public class A{}


 //sačuvaj kao B.java
   package company;
   public class B{}
```

22. Modifikatori pristupa u Javi

Postoje dva tipa modifikatora u javi: **pristupni modifikatori** i **ne-pristupni modifikatori**.

Pristupni modifikatori u javi specificiraju dostupnost (scope) podataka-članova, metoda, konstruktora ili klase.

Postoje 4 tipa java modifikatora pristupa:

1. private
2. default
3. protected
4. public

Postoji mnogo ne-pristupnih modifikatora kao što su static, abstract, synchronized, native, volatile, transient itd. Ovdje ćemo razmotriti modifikatore pristupa.

1) private modifikator pristupa
Modifikator pristupa private je dostupan samo unutar klase.

22.1.1 Jednostavan primjer private modifikatora pristupa

U ovom primjeru, kreiraćemo dve klase A i Simple. A klasa sadrži private podatak-član i private metod. Pristupićemo ovim private članovima izvan klase, tako da će se javiti compile time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data); //Compile Time Error
   obj.msg(); //Compile Time Error
   }
}
```

22.2 Uloga private konstruktora

Ako učinimo bilo koji konstruktor u klasi private, nećemo moći kreirati instancu te klase izvan te klase. Na primjer:

```
class A{
private A(){} //private konstruktor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A(); //Compile Time Error
 }
}
```

Napomena: Klasa ne može biti private ili protected osim ugniježdene klase.

22.2.1 default modifikator pristupa

Ako ne koristimo nikakav modifikator, to se tretira kao **default** modifikator. Default modifikator je dostupan samo unutar paketa.

22.2.2 Primjer default modifikatora pristupa

U ovom primjeru, kreiraćemo dva paketa pack i mypack. Pristupamo A klasi izvan njenog paketa, pošto A klasa nije public, njoj se ne može pristupiti izvan paketa.

```
//sačuvano od A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}

//sačuvano od B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
    A obj = new A(); //Compile Time Error
    obj.msg(); //Compile Time Error
  }
}
```

U ovom primjeru, dostupnost (scope) klase A i njenog metoda msg() je default tako da se njoj ne može pristupiti izvan paketa.

22.2.3 protected modifikator pristupa

Protected modifikator pristupa je dostupan unutar paketa i izvan paketa ali samo preko nasljeđivanja.
Protected modifikator pristupa može se primijeniti na podatak-član, metod i konstruktor. Ne može se primijeniti na klasu.

22.2.4 Primjer protected modifikatora pristupa

U ovom primjeru, kreiraćemo dva paketa pack i mypack. A klasa paketa pack je public, tako da joj se može pristupiti izvan paketa. Ali msg metod ovog paketa je deklarisan kao protected, tako da mu se može pristupiti izvan klase samo preko nasljeđivanja.

```
//sačuvano od A.java
   package pack;
   public class A{
   protected void msg(){System.out.println("Hello");}
   }

   //sačuvano od B.java
   package mypack;
   import pack.*;

   class B extends A{
     public static void main(String args[]){
      B obj = new B();
      obj.msg();
     }
   }
```

Izlaz: Hello

22.3 public modifikator pristupa

Public modifikator pristupa je dostupan svuda. On ima najširi domet među svim modifikatorima.

22.3.1 Primjer public modifikatora pristupa

```
//sačuvano od A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}

//sačuvano od B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Izlaz: Hello

22.4 Razumijevanje svih java modifikatora pristupa

Prikažimo modifikatore pristupa jednostavnom tabelom.

Modifikator pristupa	unutar klase	unutar paketa	izvan paketa samo preko podklase	izvan paketa
Private	DA	NE	NE	NE
Default	DA	DA	NE	NE
Protected	DA	DA	DA	NE
Public	DA	DA	DA	DA

22.5 Java modifikatori pristupa sa preklapanjem metoda

Ako preklapamo neki metod, preklopljeni metod (tj. deklarisan u podklasi) ne smije biti restriktivniji.

```
class A{
protected void msg(){System.out.println("Hello java");}
}

public class Simple extends A{
void msg(){System.out.println("Hello java");} //Compile Time Error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

Default modifikator je restriktivniji nego protected. Zato se javio compile time error.

Akcesori i mutatori

Poznato je da su instansni podaci uglavnom deklarisani sa vidljivošću (pristupom) private. Zbog toga, klasa obično obezbjeđuje servise za pristup i modifikovanje vrijednosti podataka.

Akcesor metod obezbjeđuje pristup samo za čitanje (read-only) određenoj vrijednosti. Isto tako, **mutator metod**, ponekad nazvan i modifikator metod, mijenja određenu vrijednost.

Generalno, ime akcesor metoda ima formu getX, gdje je X vrijednost kojoj on obezbjeđuje pristup. Isto tako, ime mutator metoda ima formu setX, gdje je X vrijednost koja se podešava. Zbog toga se ovi tipovi metoda ponekad nazivaju “geteri“ i “seteri“.

23. Enkapsulacija u Javi

Enkapsulacija u javi je *proces pakovanja koda i podataka zajedno u jedinstvenu jedinicu*, kao što se napr. različiti lijekovi mogu pomiješati u jednoj kapsuli.
Možemo kreirati potpuno enkapsuliranu klasu u javi tako što ćemo učiniti sve podatke-članove klase private. Tada možemo koristiti seter i geter metode za podešavanje i pristup podacima u njoj.

Java Bean klasa je primjer potpuno enkapsulirane klase.

23.1.1 Prednosti enkapsulacije u javi

Obezbjeđujući samo seter ili geter metod, možemo učiniti klasu **read-only ili write-only**.
Na ovaj način obezbjeđujemo k**ontrolu nad podacima**.

23.1.2 Jednostavan primjer enkapsulacije u javi

Pogledajmo jednostavan primjer enkapsulacije koji ima samo jedno polje sa svojim seter i geter metodima.

```
//sačuvaj kao Student.java
package com.company;
public class Student{
private String name;

public String getName(){
return name;
}
public void setName(String name){
this.name=name
}
}

//sačuvaj kao Test.java
package com.company;
class Test{
public static void main(String[] args){
Student s=new Student();
s.setName("Petar");
System.out.println(s.getName());
}
}
```

Izlaz: Petar

24. Klasa Object u Javi

Klasa **Object** je roditeljska klasa svih klasa u javi po defaultu. Drugim riječima, ona je najviša klasa u javi.

Klasa Object je pogodna ako želimo da uputimo na bilo koji objekt čiji tip ne znamo. Primijetimo da referentna varijabla roditeljske klase može upućivati na objekt klase djeteta, što je poznato kao upcasting.

Uzmimo primjer, postoji getObject() metod koji vraća objekt ali on može biti bilo kog tipa poput Employee, Student itd., pa možemo upotrijebiti referencu klase Object da uputi na taj objekt. Na primjer:
Object obj=getObject(); //ne znamo koji objekt će biti vraćen od ovog metoda

Klasa Object obezbjeđuje neka zajednička ponašanja za sve objekte kao što su da objekt može biti poređen, kloniran, notifikovan itd.

24.1 Metodi klase Object

Klasa Object obezbjeđuje mnoge metode.
Evo nekih:

Metod	Opis
public boolean equals(Object obj)	Poredi dati objekt sa tim objektom.
protected Object clone() throws CloneNotSupportedException	Kreira i vraća tačnu kopiju (klon) tog objekta.
public String toString()	Vraća string reprezentaciju tog objekta.
public final void notify()	Budi jednu nit (thread), čeka na monitor tog objekta.
public final void notifyAll()	Budi sve niti, čeka na monitor tog objekta.
public final void wait(long timeout)throws InterruptedException	Uzrokuje da tekuća nit čeka određen broj milisekundi, dok druga nit notificira (poziva notify() ili notifyAll() metod).
public final void wait(long timeout,int nanos)throws InterruptedException	Uzrokuje da tekuća nit čeka određen broj milisekundi i nanosekundi, dok druga nit notificira (poziva notify() ili notifyAll() metod).
public final void wait()throws InterruptedException	Uzrokuje da tekuća nit čeka, dok druga nit notificira (poziva notify() ili notifyAll() metod).
protected void finalize()throws Throwable	Poziva ga tzv. sakupljač smeća (garbage collector) prije nego što je objekt pokupljen.

25. Kloniranje objekta u Javi

Kloniranje objekta je način za kreiranje egzaktne kopije objekta. Za ovu svrhu koristi se metod klase Object clone().

java.lang.Cloneable interfejs mora biti implementiran od klase čiji klon objekta želimo da kreiramo. Ako ne implementiramo Cloneable interfejs, clone() metod generiše **CloneNotSupportedException**.

clone() metod je definisan u klasi Object. Sintaksa clone() metoda je sljedeća:
protected Object clone() throws CloneNotSupportedException

25.1.1 Zašto koristiti clone() metod ?

clone() metod oslobađa od dodatnog procesnog zadatka za kreiranje egzaktne kopije objekta. Ako to izvodimo koristeći ključnu riječ new, to će zahtijevati dosta procesiranja i zato koristimo kloniranje objekta.

Prednosti kloniranja objekta

Manje procesnih zadataka.

25.1.2 Primjer clone() metoda (Kloniranje objekta)

Pogledajmo jednostavan primjer kloniranja objekta

```
class Student18 implements Cloneable{
int rollno;
String name;

Student18(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

public Object clone()throws CloneNotSupportedException{
return super.clone();
}

public static void main(String args[]){
try{
Student18 s1=new Student18(101,"Petar");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
```

```
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}
 }
}
```

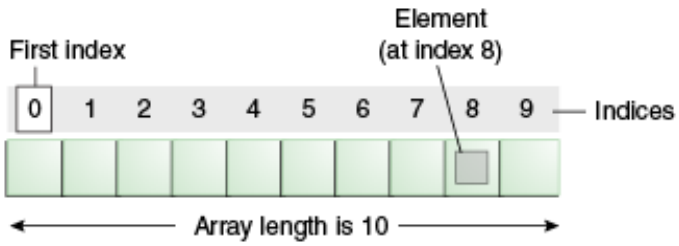
Izlaz: 101 Petar
101 Petar

Kao što se može vidjeti u ovom primjeru, obe referentne varijable imaju istu vrijednost. Stoga, clone() kopira vrijednosti jednog objekta u drugi. Tako nije potrebno pisati eksplicitni kod da kopiramo vrijednosti jednog objekta u drugi.

Ako kreiramo drugi objekt pomoću ključne riječi new i dodijelimo vrijednosti drugog objekta tom objektu, to će zahtijevati mnogo procesiranja na tom objektu. Tako da se oslobodimo dodatnog procesiranja koristimo clone() metod.

26. Java niz (Array)

Kao što je poznato, niz (array) je kolekcija elemenata sličnog tipa koja ima kontinuiranu lokaciju u memoriji.
Java niz (array) je objekt koji sadrži elemente sličnog tipa podataka. To je struktura podataka gdje spremamo slične elemente. U java niz možemo spremiti samo fiksni skup elemenata.
Niz u javi je indeksno zasnovan, prvi element niza je spremljen na indeksu 0.



26.1.1 Prednosti Java niza

- **Optimizacija koda:** čini kod optimizovanim, možemo lako pristupiti podacima ili ih sortirati.
- **Slučajni pristup:** Možemo dobiti bilo koji podatak lociran na bilo kojoj indeksnoj poziciji.

26.1. 2 Nedostaci Java niza

- **Ograničena veličina:** U niz možemo spremiti samo fiksnu količinu elemenata. Njegova veličina ne raste u vremenu izvršavanja. Da bi se riješio ovaj problem, u javi se koristi tzv. collection framework.

Tipovi nizova u javi

Postoje dva tipa niza.

- Jednodimenzionalni niz
- Višedimenzionalni niz

26.1. 3 Jednodimenzionalni niz u javi

Sintaksa deklarisanja niza u javi

dataType[] arr; (ili)
dataType []arr; (ili)
dataType arr[];

Instancijacija niza u javi

arrayRefVar=new datatype[size];

Primjer jednodimenzionalnog java niza
Pogledajmo jednostavan primjer java niza, gdje ćemo deklarisati, instancirati, inicijalizovati i prolaziti kroz niz.

```
class Testarray{
public static void main(String args[]){

int a[]=new int[5]; //deklaracija i instancijacija
a[0]=10; //inicijalizacija
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;

//ispisivanje niza
for(int i=0;i<a.length;i++)  //length (dužina) je osobina niza
System.out.println(a[i]);

}}
```

Izlaz: 10
20
70
40
50

26.1.4 Deklaracija, instancijacija i inicijalizacija java niza

Možemo zajedno deklarisati, instancirati i inicijalizovati java niz ovako:

```
int a[]={33,3,4,5};  //deklaracija, instancijacija i inicijalizacija
```

Pogledajmo jednostavan primjer koji ispisuje ovaj niz.

```
class Testarray1{
public static void main(String args[]){

int a[]={33,3,4,5};  // deklaracija, instancijacija i inicijalizacija

// ispisivanje niza
for(int i=0;i<a.length;i++)  // length (dužina) je osobina niza
System.out.println(a[i]);

}}
```

Izlaz:33
3
4
5

26.1.5 Prosljeđivanje niza metodu u javi

Možemo proslijediti java niz metodu tako da možemo ponovo upotrijebiti istu logiku na bilo koji niz.
Pogledajmo jednostavan primjer nalaženja najmanjeg broja u nizu pomoću metoda.

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];

System.out.println(min);
}

public static void main(String args[]){

int a[]={33,3,4,5};
min(a);  //prosljeđivanje niza metodu

}}
```

Izlaz:3

26.2 Višedimenzionalni niz u javi

U ovom slučaju, podaci su pohranjeni po indeksu na osnovu reda i kolone (takođe poznato kao matrična forma).

Sintaksa deklarisanja višedimenzionalnog niza u javi

dataType[][] arrayRefVar; (ili)

dataType [][]arrayRefVar; (ili)

dataType arrayRefVar[][]; (ili)

dataType []arrayRefVar[];

26.2.1 Primjer instanciranja višedimenzionalnog niza u javi

```
int[][] arr=new int[3][3];  //3 reda i 3 kolone
```

26.2.2 Primjer inicijalizovanja višedimenzionalnog niza u javi

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

26.2.3 Primjer višedimenzionalnog java niza

Pogledajmo jednostavan primjer deklarisanja, instanciranja, inicijalizovanja i ispisivanja dvodimenzionalnog niza.

```
class Testarray3{
public static void main(String args[]){

//deklarisanje i inicijalizovanje 2D niza
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//ispisivanje 2D niza
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}

}}
```

Izlaz: 1 2 3
2 4 5
4 4 5

26.2.4 Koje je ime klase java niza?

U javi, niz (array) je objekt. Za array objekt, kreirana je proxy klasa čije se ime može dobiti pomoću getClass().getName() metoda na objekt.

```
class Testarray4{
public static void main(String args[]){

int arr[]={4,4,5};

Class c=arr.getClass();
String name=c.getName();

System.out.println(name);
 }}
```

Izlaz: I

26.2.5 Kopiranje java niza

Možemo kopirati jedan niz u drugi pomoću metoda arraycopy klase System.

Sintaksa arraycopy metoda

```
public static void arraycopy(
Object src, int srcPos,Object dest, int destPos, int length
)
```

26.2.6 Primjer arraycopy metoda

```
class TestArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
            'i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];

      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
   }
}
```

Izlaz: caffein

26.2.7 Sabiranje 2 matrice u javi

Pogledajmo jednostavan primjer sabiranja dve matrice.

```
class Testarray5{
public static void main(String args[]){
//kreiranje dve matrice
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//kreiranje druge matrice u koju se sprema suma dve matrice
int c[][]=new int[2][3];

//sabiranje i ispis zbira 2 matrice
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println(); //novi red
}

}}
```

Izlaz: 2 6 8
6 8 10

27 Poziv po vrijednosti u javi (Call by Value)

U javi postoji samo poziv po vrijednosti, ne i poziv po referenci. Ako pozovemo metod koji prosljeđuje vrijednost, to je poznato kao poziv po vrijednosti. Promjene se dešavaju u pozvanom metodu, ne obuhvataju pozivajući metod.

Primjer poziva po vrijednosti u javi

U slučaju poziva po vrijednosti izvorna vrijednost nije promijenjena. P
ogledajmo jednostavan primjer:

```
class Operation{
 int data=50;

 void change(int data){
 data=data+100; //promjene će biti samo u lokalnim varijablama
 }

 public static void main(String args[]){
   Operation op=new Operation();

   System.out.println("before change "+op.data);
   op.change(500);
   System.out.println("after change "+op.data);

 }
}
```

```
Izlaz: before change 50
       after change 50
```

Primjer 2 poziva po vrijednosti u javi

U slučaju poziva po referenci izvorna vrijednost se mijenja ako mi načinimo promjene u pozvanom metodu. Ako proslijedimo objekt umjesto neke primitivne vrijednosti, izvorna vrijednost će biti promijenjena. U ovom primjeru prosljeđujemo objekt kao vrijednost. Pogledajmo jednostavan primjer:

```
class Operation2{
 int data=50;
 void change(Operation2 op){
 op.data=op.data+100; //promjene će biti u instansnoj varijabli
 }
 public static void main(String args[]){
   Operation2 op=new Operation2();
   System.out.println("before change "+op.data);
   op.change(op);  //prosljeđuje objekt
   System.out.println("after change "+op.data);
 }
}
```

```
Izlaz: before change 50
       after change 150
```

28.Ključna riječ strictfp

Ključna riječ strictfp obezbjeđuje da ćemo dobiti isti rezultat na svakoj platformi ako izvodimo operacije sa varijablom u pokretnom zarezu (floating-point). Tačnost može da se razlikuje od platforme do platforme, pa je stoga java obezbijedila ključnu riječ strictfp, tako da se na svakoj platformi može dobiti isti rezultat. Ovo daje bolju kontrolu nad aritmetikom u pokretnom zarezu.

28.1.1 Legalni kod za ključnu riječ strictfp

Ključna riječ strictfp može se primijeniti na metode, klase i interfejse.

```
strictfp class A{}  //strictfp primijenjen na klasu

strictfp interface M{}  //strictfp primijenjen na interfejs

class A{
void m(){}  //strictfp primijenjen na metod
}
```

28.1.2 Ilegalni kod za ključnu riječ strictfp

Ključna riječ strictfp se ne može primijeniti na apstraktne metode, varijable ili konstruktore.

```
class B{
strictfp abstract void m();  //Ilegalna kombinacija modifikatora
}

class B{
strictfp int data=10;  //modifikator strictfp ovdje nije dozvoljen
}

class B{
strictfp B(){}  //modifikator strictfp ovdje nije dozvoljen
}
```

29. Kreiranje API dokumenta

Moguće je kreirati API dokument u javi uz pomoć **javadoc** alata. U java fajlu, moramo koristiti dokumentacioni komentar /**... */ da postavimo informaciju o klasi, metodu, konstruktoru, poljima itd.

Pogledajmo jednostavnu klasu koja sadrži dokumentacioni komentar.

```
package com.abc;
/** Ova klasa je korisnički definisana klasa koja sadrži metod cube.*/
public class M{

/** Metod cube ispisuje kub datog broja */
public static void  cube(int n){System.out.println(n*n*n);}
}
```

Da bi se kreirao dokument API, potrebno je koristiti javadoc alat praćen imenom java fajla. Nema potrebe kompajlirati java fajl.

U komandnom promptu, treba napisati:
javadoc M.java
da bi se generisao dokument API. Biće kreiran određen broj html fajlova. Treba otvoriti index.html fajl da bi se dobile informacije o klasama.

30. Argumenti komandne linije u javi

Argument komandne linije u javi je argument proslijeđen u vrijeme izvršavanja java programa.

Argumenti proslijeđeni iz konzole mogu biti primljeni u java program i mogu biti korišteni kao ulaz.
Dakle, ovo obezbjeđuje pogodan način za provjeravanje ponašanja programa za različite vrijednosti. Možemo proslijediti **N** (1,2,3 itd.) argumenata iz komandnog prompta.

30.1.1 Jednostavan primjer argumenta komandne linije u javi

U ovom primjeru, primamo samo jedan argument i ispisujemo ga. Da pokrenemo ovaj java program, moramo proslijediti najmanje jedan argument iz komandnog prompta.

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Prvi argument je: "+args[0]);
}
}
compile by > javac CommandLineExample.java
run by > java CommandLineExample Petar
Izlaz: Prvi argument je: Petar
```

30.1.2 Primjer argumenta komandne linije koji ispisuje sve vrijednosti

U ovom primjeru, ispisujemo sve argumente proslijeđene iz komandne linije. Za ovu svrhu, prolazimo niz pomoću for petlje.

```
class A{
public static void main(String args[]){

for(int i=0;i<args.length;i++)
System.out.println(args[i]);
}
}
compile by > javac A.java
run by > java A Petar Marko 1 3 abc

Izlaz: Petar
        Marko
       1
       3
       abc
```

31. Razlika između objekta i klase

Postoje mnoge razlike između objekta i klase. Lista tih razlika je data u sljedećoj tabeli:

Br.	Objekt	Klasa
1)	Objekt je **instanca** klase.	Klasa je **nacrt ili šablon** iz kojega se objekti kreiraju.
2)	Objekt je **entitet iz realnog svijeta** kao što je olovka, laptop, mobilni telefon, krevet, tastatura, miš, stolica itd.	Klasa je **grupa sličnih objekata**.
3)	Objekt je **fizički** entitet.	Klasa je **logički** entitet.
4)	Objekt se uglavnom kreira pomoću **new ključne riječi** napr. Student s1=new Student();	Klasa je deklarisana pomoću **class ključne riječi** napr. class Student{}
5)	Objekt se kreira **mnogo puta** po potrebi.	Klasa se deklariše **jednom**.
6)	Objekt **alocira memoriju kada je kreiran**.	Klasa **ne alocira memoriju kada je kreirana**.
7)	Postoje **mnogi načini za kreiranje objekta** u javi kao što su new ključna riječ, newInstance() metod, clone() metod, factory metod i deserijalizacija.	Postoji samo **jedan način za definisanje klase** u javi pomoću ključne riječi class.

32. Razlika između preopterećenja metoda i preklapanja metoda u javi

Postoje mnoge razlike između preopterećenja metoda i preklapanja metoda u javi. Lista tih razlika je data u sljedećoj tabeli:

Br.	Preopterećenje metoda	Preklapanje metoda
1)	Preopterećenje metoda se koristi *da poboljša čitljivost* programa.	Preklapanje metoda se koristi *da obezbijedi specifičnu implementaciju* metoda koji je već obezbijeđen od njegove superklase.
2)	Preopterećenje metoda se izvodi *unutar klase*.	Preklapanje metoda se javlja *u dve klase* koje imaju IS-A (nasljeđivanje) relaciju.
3)	U slučaju preopterećenja metoda, *parametar mora biti različit*.	U slučaju preklapanja metoda, *parametar mora biti isti*.
4)	Preopterećenje metoda je primjer *compile time polimorfizma*.	Preklapanje metoda je primjer *run time polimorfizma*.
5)	U javi, preopterećenje metoda se ne može izvesti samo promjenom return tipa metoda. *Return tip može biti isti ili različit* u preopterećenju metoda. Ali neophodno je promijeniti parametar.	*Return tip mora biti isti ili kovarijantan* u preklapanju metoda.

Primjer preopterećenja metoda u javi

```
class OverloadingExample{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
```

Primjer preklapanja metoda u javi

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bone...");}
}
```

33. Metod toString() u javi

Ako želimo predstaviti bilo koji objekt kao string, **toString() metod** je veoma pogodan. Metod toString() vraća string reprezentaciju objekta.
Ako ispisujemo neki objekt, java kompajler interno poziva toString() metod za taj objekt. Tako preklapanje toString() metoda, vraća željeni izlaz, to može biti stanje nekog objekta i sl. zavisno od naše implementacije.

33.1.1 Prednosti java toString() metoda

Preklapanjem toString() metoda Object klase, možemo vratiti vrijednosti objekta, tako da nije potrebno pisati mnogo koda.

33.1.2 Razumijevanje problema bez toString() metoda

Pogledajmo jednostavan kod koji ispisuje referencu.

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public static void main(String args[]){
   Student s1=new Student(101,"Petar","Beograd");
   Student s2=new Student(102,"Marko","Banjaluka");

   System.out.println(s1);  //kompajler ovdje piše s1.toString()
   System.out.println(s2);  // kompajler ovdje piše s2.toString()
 }
}
I
```

Izlaz: Student@1fee6fc
Student@1eed786

Kao što se može vidjeti u ovom primjeru, ispis s1 i s2 ispisuje hashcode vrijednosti objekata, a mi želimo da ispisuje vrijednosti ovih objekata. Pošto java kompajler interno poziva toString() metod, preklapanje ovog metoda će vratiti specificirane vrijednosti. Razmotrimo to u sljedećem primjeru:

33.1.3 Primjer java toString() metoda

Pogledajmo sada realni primjer toString() metoda.

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public String toString(){  //preklapanje toString() metoda
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Petar","Beograd");
   Student s2=new Student(102,"Marko","Banjaluka");

   System.out.println(s1);  //kompajler ovdje piše s1.toString()
   System.out.println(s2);  // kompajler ovdje piše s2.toString()
 }
}
```

Izlaz:101 Petar Beograd
102 Marko Banjaluka

34. Klasa Java Scanner

Postoje razni načini za učitavanje ulaza sa tastature, java.util.Scanner klasa je jedan od njih. **Java Scanner** klasa dijeli ulaz u tokene koristeći ograničavač (delimiter) koji je po defaultu blank. Ona obezbjeđuje mnoge metode za čitanje i parsiranje različitih primitivnih vrijednosti.
Java Scanner klasa se mnogo koristi za parsiranje teksta za string i primitivne tipove pomoću regularnih izraza.
Java Scanner klasa nasljeđuje Object klasu i implementira Iterator i Closeable interfejse.

34.1.1 Često korišteni metodi Scanner klase

Ovo je lista često korištenih metoda klase Scanner:

Metod	Opis
public String next()	vraća sljedeći token iz scanner-a.
public String nextLine()	pomjera poziciju scanner-a na sljedeći red i vraća vrijednost kao string.
public byte nextByte()	skenira sljedeći token kao byte.
public short nextShort()	skenira sljedeći token kao short vrijednost.
public int nextInt()	skenira sljedeći token kao int vrijednost.
public long nextLong()	skenira sljedeći token kao long vrijednost.
public float nextFloat()	skenira sljedeći token kao float vrijednost.
public double nextDouble()	skenira sljedeći token kao double vrijednost.

34.1.2 Primjer Java Scanner-a za dobijanje ulaza sa konzole

Pogledajmo jednostavan primjer Java Scanner klase koja čita int, string i double vrijednost kao ulaz:

```
import java.util.Scanner;
class ScannerTest{
 public static void main(String args[]){
   Scanner sc=new Scanner(System.in);
```

```
  System.out.println("Enter your rollno");
  int rollno=sc.nextInt();
  System.out.println("Enter your name");
  String name=sc.next();
  System.out.println("Enter your fee");
  double fee=sc.nextDouble();
  System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
  sc.close();
 }
}
```

Izlaz:

```
Enter your rollno
111
Enter your name
Petar
Enter your fee
450000
Rollno:111 name: Petar fee: 450000
```

34.1.3 Primjer Java Scanner-a sa delimiterom

Pogledajmo primjer Scanner klase sa delimiterom. \s predstavlja blank.

```
import java.util.*;
public class ScannerTest2{
public static void main(String args[]){
    String input = "10 tea 20 coffee 30 tea biscuits";
    Scanner s = new Scanner(input).useDelimiter("\\s");
    System.out.println(s.nextInt());
    System.out.println(s.next());
    System.out.println(s.nextInt());
    System.out.println(s.next());
    s.close();
}}
```

Izlaz:

```
10
tea
20
coffee
```

Mala kompjuteska enciklopedija
MKE

Java

Banja Luka, 2023

Materijal prikupio i klasifikovao Duško Milinčić