Mobilne Aplikacije

Nositelj: doc. dr. sc. Nikola Tanković
Izvođač: dr. sc. Robert Šajina
Asistent: mag. inf. Alesandro Žužić

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli

Fakultet informatike u Puli

[1] Java Ponavljanje

Posljednje ažurirano: 23. listopada 2025.

Sadržaj

---

1. Osnove

Main Klasa

Klasa koja sadrži `main(String[] args)` metodu – ulazna točka programa, koristi se za pokretanje programa:

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

Za pokretanje datoteke koristi se naredba `java`:

```
java Main.java
```

Output:

```
Hello, World!
```

- **Metoda** `main(String[] args)` je obavezna u svakom Java programu. To je mjesto gdje program započinje s izvršavanjem

Sintaksa

Ako datoteka sadrži klasu `Main` onda se i ta datoteka mora nazivati `Main.java`.

- **Java razlikuje velika i mala slova**. `Main` i `main` se tretiraju kao dva potpuno različita naziva.

Za ispis u konzolu koristimo metodu `println()`:

```
System.out.println("Hello World");
// alternativno: System.out.print("Hello World");
```

- `System` je ugrađena Java klasa
- `out` (*output*) je član klase System
- `println()` (*print line*) je metoda

> Možemo alternativno koristiti `print()` metodu, u tom slučaju ne ubacuje novi redak na kraj ispisa

> **Svaka naredba** mora završiti **delimiterom** `;` (*točkom-zarezom*) !!!

Komentare možemo pisati u jednoj ili više linija:

- **jednolinijski** komentar se piše sa `// komentar`
- **višelinijski** komentar se piše sa `/* komentar */`

```
// jendolinijski komentar

/*
    Višelinijski
    komentar
*/
```

Varijable

Varijable su spremnici za pohranu vrijednosti podataka. Java definira sljedeće vrste varijabli:

- **Instance variables (non-static fields)** Varijable koje pripadaju pojedinom objektu. Njihove vrijednosti su različite za svaki objekt. Primjer: `trenutnaBrzina` kod različitih automobila
- **Class variables (static fields)** Varijable označene s `static`. Postoji samo jedna kopija varijable za sve instance klase. Primjer: `static int brojBrzina = 6;`. Moguće je dodati `final` da vrijednost ne može biti promijenjena
- **Lokalne varijable** Varijable deklarirane unutar metoda, vidljive samo unutar te metode. Primjer: `int brojač = 0;`
- **Parametri** Varijable koje metode ili konstruktori primaju kao ulaz. Primjer: `String[] args` u `main` metodi

Primarni (*primitivni*) tipovi podataka

- `byte` – pohranjuje male cijele brojeve od -128 do 127

```
byte broj = 100;
```

- `short` – pohranjuje cijele brojeve od $-32,768$ do $32,767$

```
short godina = 2025;
```

- `int` – pohranjuje cijele brojeve, od -2^{31} do 2^{31}

```
int brojStudenata = 120;
```

- `long` – pohranjuje cijele brojeve, od -2^{63} do 2^{63}

```
long populacija = 7800000000L;
```

- `float` – pohranjuje brojeve s pomičnim zarezom (*decimalne brojeve*), *single-precision 32-bit*

```
float temperatura = 12.4f;
```

- `double` – pohranjuje brojeve s pomičnim zarezom (*decimalne brojeve*) - *double-precision 64-bit*

```
double cijena = 25.98;
```

- `char` – pohranjuje pojedinačne znakove, npr. `'a'` ili `'B'`, *16-bit Unicode*

```
char grupa = 'B';
```

- `boolean` – pohranjuje dvije moguće vrijednosti: `true` ili `false`

```
boolean aktivan = true;
```

Referentni (*objektni*) tipovi podataka

- `String` – pohranjuje tekstualne podatke (*niz znakova*)

```
// Deklaracija i inicijalizacija
String ime = "Ime";
String prezime = "Prezime";

// Spajanje stringova (concatenation)
String punoIme = ime + " " + prezime;
System.out.println(punoIme); // output: Ime Prezime

// Duljina stringa
int duljina = punoIme.length();
System.out.println(duljina); // output: 11

// Dohvaćanje znaka po indeksu
char prvoSlovo = punoIme.charAt(0);
System.out.println(prvoSlovo); // output: I

// Pretvorba u velika/mala slova
System.out.println(punoIme.toUpperCase()); // output: IME PREZIME
System.out.println(punoIme.toLowerCase()); // output: ime prezime

// Provjera sadržaja stringa
System.out.println(punoIme.contains("Prezime")); // output: true
System.out.println(punoIme.startsWith("Ime")); // output: true
System.out.println(punoIme.endsWith("zime")); // output: true
```

- **Polje (Array)**
  - Polja su strukture podataka koje pohranjuju više vrijednosti istog tipa u jednom objektu
  - Svaka vrijednost u polju pristupa se putem indeksa, počevši od 0

```
// Deklaracija praznog polja s veličinom
int[] brojevi = new int[5]; // polje s 5 cijelih brojeva

// Deklaracija i inicijalizacija s vrijednostima
int[] brojevi2 = {1, 2, 3, 4, 5};

// Pristup elementima
int prvi = brojevi2[0]; // prvi element: 1
brojevi2[2] = 10;       // treći element postaje 10

// Petlja kroz polje
for (int i = 0; i < brojevi2.length; i++) {
    System.out.println(brojevi2[i]); // ispisuje sve elemente polja
}
for (int b : brojevi) {
    System.out.println(b);
}

// Multidimenzionalno polje
int[][] matrica = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println(matrica[1][2]); // output: 6
```

- **Klase (Classes)** – korisnički definirani tipovi podataka

```
class Student {
    String ime;
    int godina;
}
```

- **Objekti (Objects)** – primjer objekta

```
Student s1 = new Student();
s1.ime = "Ana";
s1.godina = 3;
```

Operatori

- **Operatori dodjele (Assignment Operators)**
  - Koriste se za dodjeljivanje vrijednosti varijablama (`=`, `+=`, `-=`, `*=`, `/=`, `%=`)

```
int a = 5;
```

- **Aritmetički operatori (*Arithmetic Operators*)**
  - Izvršavaju matematičke operacije
  - `+` zbrajanje, `-` oduzimanje, `*` množenje, `/` dijeljenje, `%` ostatak pri dijeljenju

```
int a = 10;
int b = 3;

System.out.println(a + b); // 10 + 3 = 13
System.out.println(a - b); // 10 - 3 = 7
System.out.println(a * b); // 10 * 3 = 30
System.out.println(a / b); // 10 / 3 = 3
System.out.println(a % b); // 10 % 3 = 1
```

- **Unarni operatori (*Unary Operators*)**
  - Djeluju na jedan operand.
  - `+` pozitivan, `-` negativan, `++` povećanje za 1, `--` smanjenje za 1, `!` logička negacija

```
int a = 5;
a++;
boolean b = true;
System.out.println(!b); // false
```

- **Operatori jednakosti (*Equality Operators*)**
  - Uspoređuju dvije vrijednosti.
  - `==` jednako, `!=` nije jednako

```
int a = 5;
int b = 3;
System.out.println(a == b); // false
System.out.println(a != b); // true
```

- **Relacijski operatori (*Relational Operators*)**
  - Uspoređuju veličinu dviju vrijednosti.
  - `<` manje od, `>` veće od, `<=` manje ili jednako, `>=` veće ili jednako

```
System.out.println(a > b); // true
System.out.println(a <= b); // false
```

- **Logički operatori (*Conditional/Logical Operators*)**
  - Kombiniraju logičke izraze.
  - `&&` i (*AND*), `||` ili (*OR*)

```
boolean x = true;
boolean y = false;
System.out.println(x && y); // false
System.out.println(x || y); // true
```

Kontrolne strukture (*Control Flow Statements*)

- **`if-then`** i **`if-then-else`**
  - Omogućuju izvršavanje koda ovisno o uvjetu.

```
int broj = 10;

if (broj > 5) {
    System.out.println("Broj je veći od 5"); // output: Broj je veći
od 5
} else {
    System.out.println("Broj je manji ili jednak 5");
}
```

- `switch`
  - Omogućuje izbor između više opcija temeljenih na vrijednosti izraza.

```
int dan = 3;
switch(dan) {
    case 1:
        System.out.println("Ponedjeljak");
        break;
    case 2:
        System.out.println("Utorak");
        break;
    case 3:
        System.out.println("Srijeda"); // output: Srijeda
        break;
    default:
        System.out.println("Nepoznat dan");
}
```

- `while` i `do-while`
  - Ponavljaju blok koda dok je uvjet istinit.

```
int i = 1;
while (i <= 3) {
    System.out.println(i); // output: 1 2 3
    i++;
}

int j = 1;
do {
    System.out.println(j); // output: 1 2 3
    j++;
} while (j <= 3);
```

- `for`
  - Standardna petlja s inicijalizacijom, uvjetom i inkrementom/dekrementom.

```
for (int k = 1; k <= 3; k++) {
    System.out.println(k); // output: 1 2 3
}
```

- **Grananje (`break`, `continue`, `return`)**
  - `break` – prekida petlju ili switch
  - `continue` – preskače trenutnu iteraciju petlje
  - `return` – izlazi iz metode

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue; // preskače 3
    if (i == 5) break;    // prekida petlju na 5
    System.out.println(i); // output: 1 2 4
}

int suma(int a, int b) {
    return a + b; // vraća rezultat i izlazi iz metode
}
```

---

2. OOP (Objektno-Orijentirano Programiranje)

Proceduralno programiranje se temelji na pisanju **procedura** ili metoda koje izvršavaju operacije nad podacima, dok objektno-orijentirano programiranje (*OOP*) stvara **objekte** koji sadrže i podatke i metode.

Prednosti OOP-a u odnosu na proceduralno programiranje:

- OOP je brži i lakši za izvršavanje
- OOP pruža jasnu strukturu programa
- OOP pomaže održati Java kôd **DRY** ("Don't Repeat Yourself" – Ne ponavljaj se"), što čini kôd lakšim za održavanje, izmjene i otklanjanje pogrešaka
- OOP omogućuje stvaranje **potpuno ponovljivih i višekratno upotrebljivih aplikacija** s manje kôda i kraćim vremenom razvoja

> Princip "Don't Repeat Yourself" (*DRY*) znači smanjiti ponavljanje kôda. Zajednički kôd aplikacije treba izdvojiti na jedno mjesto i ponovno koristiti umjesto da se duplicira.

Klasa

- Klasa je **predložak** (*template*) prema kojem se stvaraju objekti
- U njoj definiramo **atribute** (*podatke koje objekt pamti*) i **metode** (*ponašanja, tj. što objekt može raditi*)
- Klasa sama po sebi **nije objekt**, već opis kako će objekt izgledati i što će moći raditi
- Svi objekti iste klase imaju **istu strukturu** (*iste atribute i metode*), ali svaka instanca ima **vlastite vrijednosti atributa**

Primjer:

```
class Kolegij {
    // Atributi – varijable objekta
    String naziv;
    int ects;

    // Metoda – funkcije objekta
    void ispisiOpis() {
        System.out.println(naziv + " - " + ects + " ECTS");
    }
}
```

Objekt

- Objekt je **konkretna instanca klase**, stvarni entitet koji zauzima memoriju
- Kad kreiramo objekt pomoću ključne riječi `new`, u memoriji se stvara prostor za njegove atribute i metode
- Svaki objekt ima **vlastite vrijednosti** atributa

Primjer:

```
class Main {
  public static void main(String[] args) {
      System.out.println("Hello, World!");

      Kolegij kol_1 = new Kolegij(); // prvi objekt
      Kolegij kol_2 = new Kolegij(); // drugi objekt

      kol_1.naziv = "Mobilne Aplikacije";
      kol_1.ects = 6;

      kol_2.naziv = "Programsko Inženjerstvo";
      kol_2.ects = 6;

      kol_1.ispisiOpis();
      kol_2.ispisiOpis();
  }
}
```

- `c1` i `c2` su **dvije različite instance** klase `Kolegij`
- Oba imaju iste **atribute i metode**, ali **drugačije vrijednosti** atributa (`naziv`, `ects`)
- Kad pozovemo `ispisiOpis()` nad svakim objektom, rezultat ovisi o njihovom **trenutnom stanju**

Samostalni zadatak za vježbu 1

Napiši program koji definira klasu **Automobil**. Klasa treba sadržavati sljedeće:

1. Atribute:
    - `marka` (*String*)
    - `model` (*String*)
    - `godinaProizvodnje` (*int*)
2. Metodu:
    - `ispisiPodatke()` – ispisuje sve podatke o automobilu u jednom redu
    - `starostAutomobila(int trenutnaGodina)` - računa i ispisuje koliko je automobil star
3. U klasi `Main`:
    - Stvori **tri različita objekta** tipa `Automobil`
    - Svakom objektu dodijeli različite vrijednosti atributa
    - spremite objekte u polje automobili
    - Pozovi metodu `ispisiPodatke()` za svaki objekt koristeći `for` petlju

Primjer ispisa:

```
Marka: Toyota, Model: Corolla, Godina: 2020
Marka: Ford, Model: Focus, Godina: 2018
Marka: BMW, Model: X5, Godina: 2023
```

Konstruktor i Preopterećenje (*Overloading*)

Konstruktor je posebna metoda unutar klase koja se koristi za **inicijalizaciju objekta** u trenutku kada se stvori njegova instanca.

- Konstruktor uvijek ima **isto ime kao klasa**
- Ne vraća vrijednost (nema `return`)
- Može postojati više konstruktora u istoj klasi, što nazivamo **preopterećenje konstruktora (overloading)**

Preopterećenje konstruktora znači da možemo definirati više verzija konstruktora unutar iste klase, ali se razlikuju po **broju ili tipu argumenata**.

Primjer:

```
class Kolegij {
    String naziv;
    private int ects;

    // Prazan konstruktor – kreira objekt s "default" vrijednostima
    Kolegij() {}

    // Konstruktor s argumentima – inicijalizira objekt s danim vrijednostima
    Kolegij(String naziv, int ects) {
        this.naziv = naziv;
        this.ects = ects;
    }
}
```

Pravila za preopterećenje (overloading):

1. **Broj argumenata** – dvije metode mogu imati isto ime ako primaju različit broj parametara
2. **Tip argumenata** – metode mogu imati isti broj argumenata, ali različite tipove podataka
3. **Redoslijed argumenata** – ako tipovi nisu jednaki, redoslijed također razlikuje metode

---

Samostalni zadatak za vježbu 2

Napiši program koji definira klasu **Student**.

1. Atributi klase:

- `ime` (*String*)
- `prezime` (*String*)
- `godinaUpisa` (*int*)

2. Konstruktori:

- Prazni konstruktor – postavlja sve atribute na default vrijednosti
- Konstruktor s parametrima – inicijalizira sve atribute vrijednostima koje proslijedimo

3. Metode:

- `ispisiPodatke()` – ispisuje sve podatke o studentu u jednom retku
- Preopterećena metoda `ispisiPodatke(boolean detaljno)`:
    - Ako je `detaljno = true`, ispisuje dodatne informacije (*koliko godina student studira ako se uzme trenutna godina*).
    - Ako je `detaljno = false`, ispisuje samo ime i prezime

4. U klasi Main:

- Stvori **dva objekta** klase `Student`, jedan pomoću praznog konstruktora, drugi pomoću konstruktora s parametrima
- Pozovi obje verzije metode `ispisiPodatke()` za svaki objekt

Enkapsulacija

Njena svrha je **sakriti unutarnje podatke klase** i omogućiti kontrolirani pristup tim podacima putem javnih metoda – **getter** i **setter**.

- **Privatni atributi (`private`)** – ne mogu se mijenjati izvan klase
- **Javne metode (`public get i set`)** – omogućuju siguran pristup i izmjenu atributa

Primjer:

```
public class Kolegij {
    private String naziv;
    private int ects;

    public Kolegij(String naziv, int ects) {
        this.naziv = naziv;
        this.ects = ects;
    }

    // Getter za naziv
    public String getNaziv() {
        return naziv;
    }

    // Setter za naziv
    public void setNaziv(String noviNaziv) {
        this.naziv = noviNaziv;
    }

    // Getter za ECTS
    public int getEcts() {
        return ects;
    }

    // Setter za ECTS s validacijom
    public void setEcts(int noviEcts) {
        if (noviEcts > 0) {
            this.ects = noviEcts;
        } else {
            System.out.println("ECTS mora biti pozitivan broj!");
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Kolegij kol = new Kolegij("Matematika", 6);

        // Dohvat vrijednosti preko getter-a
        System.out.println("Naziv: " + kol.getNaziv() + ", ECTS: " +
kol.getEcts());

        // Promjena vrijednosti preko setter-a
        kol.setNaziv("Fizika");
        kol.setEcts(5);

        System.out.println("Naziv: " + kol.getNaziv() + ", ECTS: " +
kol.getEcts());
    }
}
```

Samostalni zadatak za vježbu 3

Napiši program koji definira klasu **Student**.

1. Atributi (sve privatni):

- `ime` (String)
- `prezime` (String)
- `godinaUpisa` (int)

2. Konstruktor:

- Konstruktor koji inicijalizira sve atribute vrijednostima koje proslijedimo.

3. Javne metode (getter i setter):

- `getIme()` i `setIme(String novoIme)`
- `getPrezime()` i `setPrezime(String novoPrezime)`
- `getGodinaUpisa()` i `setGodinaUpisa(int novaGodina)`
    - Setter za `godinaUpisa` treba **provjeriti** da godina nije manja od 2000. Ako jest, ispiši upozorenje i ne mijenjaj vrijednost
- Setter za `ime` i `prezime` treba **provjeriti** da nisu prazni string

4. U klasi `Main`:

- Stvori **dva objekta** klase `Student`
- Pokušaj promijeniti atribute koristeći settere, uključujući nevalidne vrijednosti
- Ispiši stanje objekata koristeći gettere

Primjer:

```
Ime: Ana, Prezime: Horvat, Godina upisa: 2021
Ime: Ivan, Prezime: Ivić, Godina upisa: 2022
Godina upisa mora biti 2000 ili kasnije!
Ime ne smije biti prazno!
```

Agregacija i Kompozicija

U objektno orijentiranom programiranju klasa može koristiti objekte drugih klasa na dva načina: **kompozicijom** i **agregacijom**.

Kompozicija

- Klasa **interno stvara** objekte drugih klasa i upravlja njihovim životnim ciklusom
- Ako se objekt "vlasnika" uništi, i objekti koje sadrži obično prestaju postojati
- Odnosi se na **jaku vezu** između objekata

Primjer:

```java
import java.util.*;

class Kolegij {
    private String naziv;
    private int ects;

    public Kolegij(String naziv, int ects) {
        this.naziv = naziv;
        this.ects = ects;
    }

    public String getNaziv() {
        return naziv;
    }

    public int getEcts() {
        return ects;
    }
}
```

```java
class Student {
    private String ime;
    private List<Kolegij> kolegiji = new ArrayList<>();

    public Student(String ime) {
        this.ime = ime;
    }

    public void upisiKolegij(String nazivKolegija, int ects) {
        kolegiji.add(new Kolegij(nazivKolegija, ects));
    }

    public void ispisiKolegije() {
        for (Kolegij k : kolegiji) {
            System.out.println(k.getNaziv() + " - " + k.getEcts() + " ECTS");
        }
    }

    public String getIme() {
        return ime;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Ana");

        s.upisiKolegij("Matematika", 6);
        s.upisiKolegij("Programiranje", 5);
        s.upisiKolegij("Fizika", 4);

        System.out.println("Kolegiji studenta " + s.getIme() + ":");
        s.ispisiKolegije();
    }
}
```

- Ovdje `Student` **sam kreira** objekte `Kolegij` i upravlja njima

Agregacija

- Klasa **koristi ili referencira** objekte koji postoje izvan nje
- Ako se objekt uništi, vanjski objekti i dalje postoje
- Odnosi se na **slabiju vezu** između objekata

Primjer:

```
public void upisiKolegij(Kolegij kolegij) {
    kolegiji.add(kolegij); // Student samo dodaje referencu na postojeći
Kolegij
}
```

- Ovdje `Kolegij` objekt može postojati i bez `Student` objekta

Samostalni zadatak za vježbu 4

Napiši program koji definira klase **Knjiznica** i **Knjiga**.

1. Klasa Knjiga:

- Atributi (privatni):
  - `naslov` (String)
  - `autor` (String)
- Konstruktor koji inicijalizira sve atribute.
- Javne metode (getter i setter) za svaki atribut.

2. Klasa Knjiznica:

- Atributi (privatni):
  - `naziv` (String)
  - `knjige` (lista objekata Knjiga)
- Metode:
  - `dodajKnjigu(String naslov, String autor)` – stvara novi objekt Knjiga i dodaje ga u listu.
  - `dodajKnjigu(Knjiga knjiga)` – dodaje već postojeći objekt Knjiga u listu
  - `ispisiSveKnjige()` – ispisuje sve knjige u knjižnici (naslov i autora)

3. U klasi Main:

- Stvori knjižnicu.
- Dodaj najmanje **tri knjige** – barem jednu koristeći kompoziciju, barem jednu koristeći agregaciju
- Pozovi metodu `ispisiSveKnjige()` kako bi se prikazale sve knjige u knjižnici

Nasljeđivanje i Polimorfizam

Nasljeđivanje (*Inheritance*)

Omogućuje da jedna klasa **naslijedi atribute i metode** druge klase, čime se izbjegava dupliciranje koda i omogućuje proširenje funkcionalnosti.

- `super` se koristi u podklasi za pristup konstruktoru ili metodama nadklase, omogućujući inicijalizaciju naslijeđenih atributa i/ili pozivanje originalnog ponašanja metode roditelja.

Primjer:

```
public class Zivotinja {
    protected String ime;

    public Zivotinja(String ime) {
        this.ime = ime;
    }
    public void predstaviSe() {
        System.out.println("Ja sam " + ime);
    }
}
```

Podklase

```
public class Pas extends Zivotinja {
    public Pas(String ime) {
        super(ime);
    }
    public void zalaji() {
        System.out.println("Vau vau!");
    }
}

public class Macka extends Zivotinja {
    public Macka(String ime) {
        super(ime);
    }
    public void zamjauci() {
        System.out.println("Mjau mjau!");
    }
}
```

- `Pas` i `Macka` nasljeđuju sve atribute i metode iz klase `Zivotinja`
- Podklase mogu koristiti metodu `predstaviSe()` iz nadklase
- `super(ime)` u konstruktoru poziva konstruktor roditeljske klase `Zivotinja` kako bi se inicijalizirao atribut `ime`

Polimorfizam

Polimorfizam omogućuje da **varijabla tipa nadklase** poziva metode koje se ponašaju različito ovisno o stvarnom tipu objekta.

```
public class Zivotinja {
    protected String ime;

    public Zivotinja(String ime) {
        this.ime = ime;
    }

    public String opisZvuka() {
        return "Ova životinja proizvodi neki zvuk.";
    }
}

public class Pas extends Zivotinja {
    public Pas(String ime) {
        super(ime);
    }

    @Override
    public String opisZvuka() {
        return "Pas " + ime + " se glasa: Vau vau!";
    }
}

public class Macka extends Zivotinja {
    public Macka(String ime) {
        super(ime);
    }

    @Override
    public String opisZvuka() {
        return "Mačka " + ime + " se glasa: Mjau mjau!";
    }
}

public class Main {
    public static void main(String[] args) {
        Zivotinja z1 = new Pas("Rex");
        Zivotinja z2 = new Macka("Mica");

        Zivotinja[] zivotinje = { z1, z2 };

        for (Zivotinja z : zivotinje) {
            System.out.println(z.opisZvuka());
        }
    }
}
```

- Varijabla tipa `Zivotinja` može držati objekte različitih podklasa
- Poziv `opisZvuka()` izvršava stvarnu implementaciju metode iz podklase, što pokazuje **dinamički polimorfizam**.
- `@Override` omogućuje podklasama da nadjačaju i prilagode ponašanje metode iz nadklase

---

Modifikatori (Modifiers)

Modifikatori se koriste za kontrolu pristupa klasama, atributima, metodama i konstruktorima ili za dodavanje posebnih funkcionalnosti.

Access Modifiers

- **public** – dostupno iz svih klasa
- **private** – dostupno samo unutar klase
- **protected** – dostupno unutar iste klase i podklasa

Primjer:

```
class Osoba {
    public String ime = "Ivan";     // Public - dostupan svugdje
    private int starost = 30;       // Private - dostupan samo unutar klase
}
```

```
public class Main {
    public static void main(String[] args) {
        Osoba p = new Osoba();
        System.out.println(p.ime);        // Radi
        // System.out.println(p.starost); // Greška: private
    }
}
```

```
class Osoba {
    protected String grad = "Pula"; // Protected - dostupan unutar podklasa
}
class Student extends Osoba {
    public void ispisiGrad() {
        System.out.println("Grad: " + this.grad); // Protected atribut je
dostupan u podklasi
    }
}
public class Main {
    public static void main(String[] args) {
        Osoba o = new Osoba();
        Student s = new Student();
        s.ispisiGrad();               // Ispisuje: Grad: Pula
        //System.out.println(o.grad); // Greška: private
    }
}
```

> Ako modifikator nije naveden, Java koristi **default (*bez modifikatora pristupa*)** - dostupan **samo unutar istog paketa**

Non-Access Modifiers

- **final** – ne dopušta nadjačavanje metoda, promjenu varijable ili nasljeđivanje klase
- **static** – atribut ili metoda pripada klasi, ne objektu; može se koristiti bez kreiranja objekta
- **abstract** – koristi se u apstraktnim klasama i metodama; metoda nema tijelo, tijelo definira podklasa

Primjer:

```
// Final varijable
public class Main {
    final int x = 10;
    final double PI = 3.14;
}

// Static metoda
public class Main {
    static void myStaticMethod() {
        System.out.println("Može se pozvati bez objekta");
    }

    public void myPublicMethod() {
        System.out.println("Potrebno je stvoriti objekt");
    }

    public static void main(String[] args) {
        myStaticMethod(); // Radi
        Main obj = new Main();
        obj.myPublicMethod(); // Radi
    }
}
```

Primjer:

```
abstract class Vozilo {
    public abstract void vozi(); // nema tijela

    public void tip() {
        System.out.println("Opće vozilo");
    }
}

class Automobil extends Vozilo {
    public void vozi() { // tijelo metode definira podklasa
        System.out.println("Automobil vozi");
    }
}

public class Main {
    public static void main(String[] args) {
        Vozilo v = new Automobil();
        v.tip();  // Opće vozilo
        v.vozi(); // Automobil vozi
    }
}
```

Samostalni zadatak za vježbu 5

Stvorite sljedeće klase:

1. Klasa Sportaš (osnovna klasa)

Atributi (privatni/protected):

- `ime` (String)
- `godine` (int)

Konstruktor:

- Inicijalizira oba atributa.
- **Metode:**
  - `getIme()`, `getGodine()` – getter metode.
  - `setIme(String ime)`, `setGodine(int godine)` – setter metode sa provjerom (`godine >= 0`).
  - `oIgracu()` – metoda koja vraća ime i godine igrača
  - `opisVjezbe()` – metoda koja vraća tekst, npr. `"Sportaš izvodi neku vježbu."`

2. Podklase Sportaš

- **Nogometas** – nadjačava `opisVjezbe()` i vraća `"{ime} igra nogomet."`
- **Kosarkas** – nadjačava `opisVjezbe()` i vraća `"{ime} igra košarku."`
- **Plivac** – nadjačava `opisVjezbe()` i vraća `"{ime} pliva."`

3. Klasa Tim

- **Atributi (*privatni*)**:
  - `naziv` (*String*) – naziv tima
  - `sportasi` – lista objekata `Sportaš`
- **Metode:**
  - `dodajSportasa(Sportaš s)` – dodaje već postojećeg sportaša u listu.
  - `ispisiSveSportase()` – ispisuje sve sportaše u timu: ime, godine, opis vježbe.

4. Klasa SportskiKlub

- **Atributi:**
  - `naziv` (String)
  - `timovi` – lista objekata `Tim`
- **Metode:**
  - `dodajTim(Tim t)` – dodaje Tim u listu
  - `ispisiSveTimove()` – ispisuje sve timove i sportaše u njima

5. Klasa Main

- U `main` metodi:
  1. Stvori sportski klub.
  2. Stvori nekoliko timova.
  3. Dodaj sportaše u timove.
  4. Pozovi metodu za opis vježbe sportaša za dva sportaša.
  5. Pozovi metode za ispis svih sportaša po timovima i po klubu.

Primjer:

```
Ana pliva.
Luka igra nogomet.

Sportski klub: Istra

1. Tim: Plivački tim
-----------------
1. Ime: Ana (20)
2. Ime: Ivan (22)

2. Tim: Nogometni tim
-----------------
1. Ime: Marko (21)
2. Ime: Luka (23)
```

Generics

Generics u Javi omogućuju **definiranje klasa, metoda i kolekcija koje rade s različitim tipovima podataka** bez potrebe za kastanjem (*casting*) i bez gubitka tip-sigurnosti

- Koriste se uglate zagrade `<T>` gdje `T` predstavlja **tip koji će biti određen pri korištenju**

Primjer generičke klase

```
// Generic klasa Box koja može držati bilo koji tip objekta
class Box<T> {
    private T sadrzaj;

    public void set(T sadrzaj) {
        this.sadrzaj = sadrzaj;
    }

    public T get() {
        return sadrzaj;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Box za String
        Box<String> stringBox = new Box<>();
        stringBox.set("Hello Generics");
        System.out.println(stringBox.get());

        // Box za Integer
        Box<Integer> intBox = new Box<>();
        intBox.set(100);
        System.out.println(intBox.get());
    }
}
```

Output:

```
Hello Generics
100
```

- `Box<String>` znači da objekt `Box` sada može držati samo `String`
- `Box<Integer>` znači da objekt `Box` sada može držati samo `Integer`
- Nema potrebe za pretvaranje tipa podataka (*casting*), jer tip je sigurno definiran

---

Generic metode

Možemo napraviti i generičke metode unutar običnih ili generičkih klasa.

```
public class Main {
    // Generička metoda koja vraća istu vrijednost koju primi
    public static <T> T prikazi(T vrijednost) {
        System.out.println(vrijednost);
        return vrijednost;
    }

    public static void main(String[] args) {
        String s = prikazi("Test");    // T = String
        Integer i = prikazi(123);      // T = Integer
    }
}
```

- Sintaksa `<T>` prije tipa povratne vrijednosti označava da je metoda generička
- Tip `T` se određuje automatski pri pozivu metode

Generics i kolekcije

Generics se često koriste s kolekcijama poput `ArrayList`, `HashMap`, itd., kako bi se **izbjegao casting**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> imena = new ArrayList<>();
        imena.add("Ana");
        imena.add("Ivan");

        for (String ime : imena) {
            System.out.println(ime);
        }

        // ArrayList<Integer> radi s integerima
        ArrayList<Integer> brojevi = new ArrayList<>();
        brojevi.add(10);
        brojevi.add(20);

        for (int broj : brojevi) {
            System.out.println(broj);
        }
    }
}
```

- `ArrayList<String>` može držati samo `String` objekte
- `ArrayList<Integer>` može držati samo `Integer` objekte

Bounded Generics

Možemo ograničiti tipove koje generic može prihvatiti koristeći `extends`.

```
class Kutija<T extends Number> { // T može biti samo Number ili njegove
podklase
    private T broj;

    public void set(T broj) { this.broj = broj; }
    public T get() { return broj; }
}

public class Main {
    public static void main(String[] args) {
        Kutija<Integer> k1 = new Kutija<>();
        k1.set(10);

        Kutija<Double> k2 = new Kutija<>();
        k2.set(3.14);

        // Kutija<String> k3 = new Kutija<>(); // ❌ Greška: String ne
nasljeđuje Number
    }
}
```

- `<T extends Number>` znači da generic može prihvatiti samo `Number` i njegove podklase (`Integer`, `Double`, `Float`, itd.)
- Pomaže da se ograniči upotreba tipova i poveća sigurnost koda

---

Samostalni zadatak za vježbu 6

1. Napiši generičku klasu **Par** koja može držati dva objekta bilo kojeg tipa (`T` i `U`).
    - Metode: `getPrvi()`, `getDrugi()`, `setPrvi(T)`, `setDrugi(U)`.
2. U `Main` metodi:
    - Stvori `Par<String, Integer>` i inicijaliziraj ga
    - Ispiši oba elementa
    - Promijeni vrijednosti koristeći settere i ponovno ispiši
3. Pokušaj stvoriti `Par<Integer, String>`
    - Promijeni vrijednosti prvog para koristeći String, što će se dogoditi?