

# Osnove objektno-orijantisanog programiranja

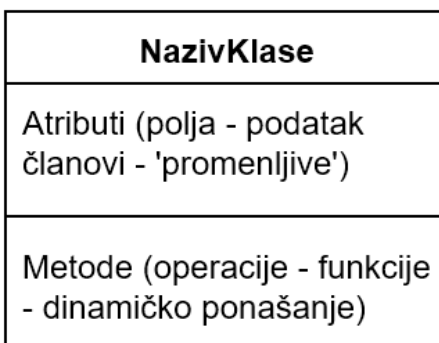
## 1. Osnovni koncepti

U objektno-orijentisanom (OO) programiranju, aplikacije se sastoje od kolekcije objekata koji međusobno interaguju. Svaki **objekat** predstavlja instancu klase, pri čemu **klasa** predstavlja šablon (recept, blueprint) koji opisuje sve karakteristike objekta. Objekat se sastoji od podataka (koji su sačuvani u poljima/atributima/svojstvima objekta), koji određuju njegovo stanje, i procedura (metoda, odnosno funkcija objekta), koje određuju njegovo ponašanje.

Klasa je apstraktan pojam koji opisuje neku grupu objekata. Klasa može da bude **stolica**, **tabla**, ili **čovjek**. Objekat predstavlja konkretnu instancu neke klase. Učionica u kom prisustvujete kursu je puna objekata čija klasa je stolica. To su, dakle, konkretne stolice koje se mogu locirati u prostoru (odnosno u memoriji, kada pričamo o računarskom programu). Tabla u računarskoj učionici 6 je jedna instanca klase tabla, dok je drugi takav objekat tabla koja je u učionici 7. Najzad, svaki polaznik, pa i predavač, se može posmatrati kao pojedinačni objekat, odnosno instanca klase čovek. U stvarnom svetu, često se sreću pojedinačni objekti vrlo sličnih karakteristika, odnosno, iste vrste. Može postojati mnogo različitih stolica, ali sve imaju zajedničke osobine, zbog toga što su pravljene po istom šablonu. Ovaj šablon predstavlja klasa.

Klasa opisuje sve objekte određenog tipa i definiše: koje osobine svi objekti tog tipa imaju (ovo nazivamo atributima/svojstvima/poljima klase); programski kod koji se može izvršiti nad objektom tog tipa (ovo nazivamo metodama klase).

U literaturi klase se predstavljaju kao pravougaonici koji se sastoje iz tri sekcije, ilustrovano na slici 1 ([UML notacija](#)):



Slika 1. UML predstavljanje klase

### 1. Naziv klase

- naziv počinje velikim slovom
- sastoji se od jedne ili više imenica u jednini koje precizno opisuju sadržaj klase
- naziv klase treba da bude jedinstven
- [Pascal case \(upper camel case\)](#)

2. Atributi (promenljive, polja, stanje)
  - vrednosti atributa definišu stanje objekta
  - kada se promeni vrednost bilo kog atributa kaže se da je objekat promenio svoje stanje
3. Metode (operacije, funkcije, ponašanja)
  - predstavljaju akcije koje definišu ponašanje objekta
  - sadrže kod koji manipuliše stanjem objekta
  - mogu imati povratnu vrednost i parametre

Primer jednostavnih klasa je dat na slici:

Radnik	Automobil
ime radnoVremeNedeljno ukupnoOdradjenoSatiNedeljno osnovica koeficijentRada	brojTablica brzina kolicinaGoriva predjenaKilometraza
povecajOdradjenoSati() izracunajPlatu() preostaloSatiNedeljno()	upali() ubrzaj() uspori() ugasi()

Slika 2. Klase Radnik i Automobil

Klasa čije ime je **radnik** sadrži attribute ime i radnoVremeNedeljno, ukupnoOdradjenoSatiNedeljno, osnovica, koeficijentRada. U realnosti, jedan radnik sadrži mnogo više atributa, uključujući adresu stanovanja, ukupan radni staž, broj slobodnih dana, itd. Klase predstavljaju model realnosti, tako da sadrže samo mali podskup podataka koji bi se mogao naći u prirodi, i to one podatke koje su bitne za rad programa, odnosno računarskog sistema za kog pišemo program.

Metode klase predstavljaju akcije koje definišu ponašanje objekta, i sadrže instrukcije na osnovu kojih se menja stanje objekta. U primeru klase radnik navedene su tri metode: povecajOdradjenoSati, izracunajPlatu, preostaloSatiNedeljno. Svaki radnik ima svoje radno vreme, odnosno broj sati koje treba da odradi u toku radne nedelje (40 sati ukoliko se radi o punom radnom vremenu), tako da je potrebno evidentirati na kraju svakog radnog dana koliko sati je radnik proveo na svom radnom mestu. Metoda povecajOdradjenoSati kao parametar prima koliko sati je radnik proveo na svom radnom mestu u toku dana, i sa tim parametrom povećava vrednosti atributa ukupnoOdradjenoSatiNedeljno. Uz pomoć metode preostaloSatiNedeljno moguće je videti koliko sati je preostalo radniku da odradi te nedelje. Ta metoda vraća razliku atributa radnoVremeNedeljno i ukupnoOdradjenoSatiNedeljno. Svaki radnik treba da dobije platu, metoda izracunajPlatu koristi attribute osnovica i koeficijenta rada da izračuna koliku platu treba da dobije radnik (u realnom sistemu bi bili korišćeni i dodatni podaci za izračunavanje plate, poput broj prekovremenih sati, iskorišćeno bolovanje i sl.).

Na slici 4 je su prikazane dve instance klase radnik, objekti **pera** i **ana**:

pera:Radnik	ana:Radnik
ime: "Pera Peric" radnoVremeNedeljno: 40 ukupnoOdradjenoSatiNedeljno: 20.5 osnovica: 600.0 koeficijentRada: 2.15	ime: "Ana Anic" radnoVremeNedeljno: 30 ukupnoOdradjenoSatiNedeljno: 25.0 osnovica: 550.0 koeficijentRada: 1.00
povacajOdradjenoSati() izracunajPlatu() preostaloSatiNedeljno()	povacajOdradjenoSati() izracunajPlatu() preostaloSatiNedeljno()

Slika 3. Instance klase Radnik

## 1.1. Definisanje klase u programskom jeziku TypeScript

U programskom jeziku TypeScript klasa se definiše upotrebom ključne reči `class`:

```
class Radnik {
    ime: string;
    radnoVremeNedeljno: number;
    ukupnoOdradjenoSatiNedeljno: number;
    osnovica: number;
    koeficijentRada: number;

    povecajOdradjenoSati(sati: number): void { ... }
    izracunajPlatu(): void { ... }
    preostaloSatiNedeljno(): number { ... }
}
```

## 1.2. Kreiranje instance klase

Za pravljenje instance klase potrebno je:

- Deklarisati identifikator instance te klase (promenljivu za objekat te klase)
- Konstruisati instancu (instancirati objekat klase, tj. alocirati memoriju za taj objekat i inicijalizovati objekat)
  - Ovo se postiže upotrebom operatora `new`
  - Operator `new` kreira objekat i vraća referencu na njega

Primer:

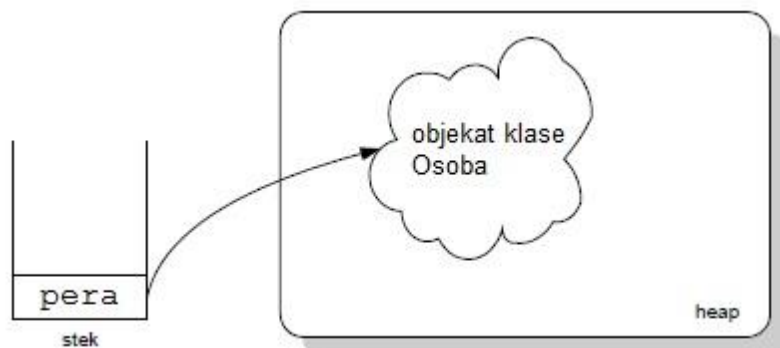
```
let pera, ana: Radnik;  
pera = new Radnik();  
ana = new Radnik();  
// ili  
let mika: Radnik = new Radnik();  
let zika: Radnik = new Radnik();
```

### 1.3. Dot (.) operator

Kada se instancira objekat klase, potreban je način da se pristupi atributima i metodama koje su propisani klasom. Za to se koristi operator . (dot). Dot operator se poziva nad objektima tj. njihovim identifikatorima. Primer:

```
let pera: Radnik = new Radnik();  
pera.ime = "Pera Peric";  
pera.ukupnoOdradjenoSatiNedeljno = 0;  
pera.povecajOdradjenoSati(15);
```

Identifikator pomoću kog se pristupa objektu (u našem slučaju pera i mika) predstavlja referencu na objekat u memoriji računara. Referenca se pridružuje objektu koji se nalazi u memoriji (heap) računara, što je i prikazano na Slici 5:



Slika 4. Referenciranje

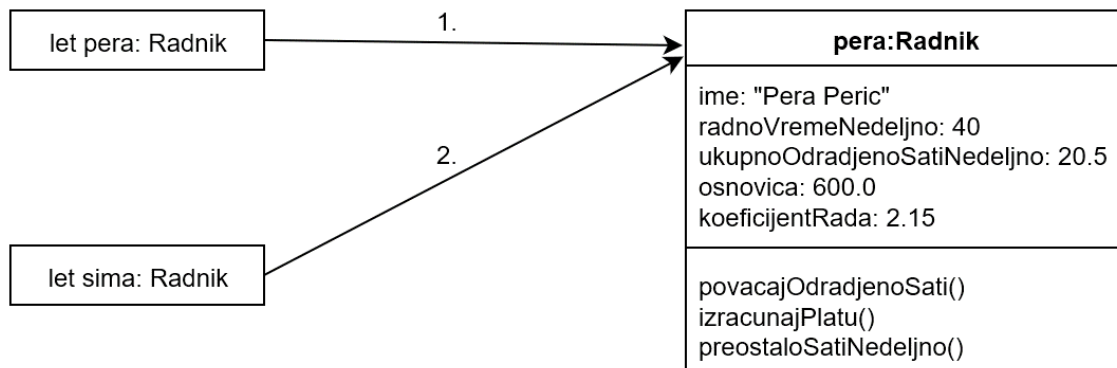
Primer rada referenci:

```
//1. Instanciranje objekta pera:Radnik
let pera: Radnik = new Radnik();
pera.ime = "Pera Peric";
pera.osnovica = 650.0;

//2. Dodela promenljivoj sima vrednost promenljive pera (reference)
let sima: Radnik = pera;

//Izmena vrednosti atributa ime u objektu
sima.ime = "Sima Simic";
```

Koja bi bila vrednost `pera.ime`? U ovom slučaju vrednost `pera.ime` bi bila „Sima Simic“. Korišćenjem ključne reči **new** (1) kreiran je novi objekat u memoriji, a referenca `pera` je postavljena tako da pokazuje na njega. Dodelom **let sima: Radnik = pera** (2) nije kreiran novi objekat u memoriji, već je vrednost reference `pera` dodeljena novokreiranoj referenci `sima`, što znači da suštinski reference `sima` i `pera` od tog trenutka pokazuju na isti objekat u memoriji. To znači da će promena objekta korišćenjem bilo koje od te dve reference uzrokovati promene nad tim jednim, zajedničkim objektom u memoriji.



Slika 5. `pera` i `sima` referenciraju isti objekat

## 1.4. Konstruktor

Konstruktor u TypeScriptu je posebna metoda koja konstruiše objekat neke klase. Za definisanje konstruktora se koristi ključna reč `constructor`. Ukoliko se konstruktor ne navede eksplicitno unutar neke klase, ta klasa će imati podrazumevani konstruktor (konstruktor koji nema parametre). U TypeScriptu moguće je imati samo jedan konstruktor u klasi.

Primer:

```

class Radnik {
    ime: string;
    radnoVremeNedeljno: number;
    ukupnoOdradjenoSatiNedeljno: number;
    osnovica: number;
    koeficijentRada: number;

    //Konsturktor bez parametara
    constructor(){
        this.ime = "";
        this.radnoVremeNedeljno = 0;
        this.ukupnoOdradjenoSatiNedeljno = 0;
        this.osnovica = 0;
        this.koeficijentRada = 0;
    }

    /*
    //Konsturktor sa parametrima
    constructor(novIme, novoRVN, novoUOSN, novaOsnov:
        this.ime = novIme;
        this.radnoVremeNedeljno = novoRVN;
        this.ukupnoOdradjenoSatiNedeljno = novoUOSN;
        this.osnovica = novaOsnovica;
        this.koeficijentRada = noviKR;
    }
    */

    povecajOdradjenoSati(sati: number): void { ... }
    izracunajPlatu(): void { ... }
    preostaloSatiNedeljno(): number { ... }
}

```

Konstruktor se može pozvati samo upotrebom operatora new, tako pozvani konstruktor instancira odgovarajući objekat.

## 1.5. Modifikatori kontrole pristupa

Modifikatori kontrole pristupa se koriste kada želimo da kontroliramo vidljivost<sup>1</sup> atributa ili metode u klasi. Navode se ispred definicija metoda i atributa.

U TypeScriptu postoje tri modifikatora kontrole pristupa:

1. **private** – atributi i metode su vidljivi samo unutar klase
2. **protected** – atributi i metode su vidljivi samo za klase naslednice
3. **public** – atributi i metode su vidljivi za sve klase u programu
  - a. Ukoliko se eksplicitno ne navede modifikator pristupa ispred atributa ili metoda, TypeScript podrazumeva da je taj atribut ili metoda public

Kada napišemo klasu i odredimo koji atributi i metode će imati koji modifikator pristupa, dobijamo kod koji se može upotrebiti u drugim klasama ili programima. Pomoću javnih (public) metoda objekti mogu međusobno da komuniciraju bez znanja o tome kako je nešto implementirano u metodi (detalji implementacije su skriveni, tj. enkapsulirani unutar klase).

U UML dijagramima public članovi su obeleženi sa prefiksom „+“, private članovi sa prefiskom „-“, i protected članovi sa prefiskom „#“, npr.:

Radnik
-ime -radnoVremeNedeljno -ukupnoOdradjenoSatiNedeljno -osnovica -koeficijentRada
+povacajOdradjenoSati() +izracunajPlatu() +preostaloSatiNedeljno()

Slika 6. Klasa sa modifikatorima kontrole pristupa

Na gore navedenoj slici za private članove klase obeleženi su atributi klase (ime, radnoVremeNedeljno i sl), a metode su obeležene sa public modifikatorom. Što znači da, u ovom konkretnom slučaju, ne bismo mogli direktno da pristupam atributima sa kodom koji se nalazi izvan klase. Kod klase:

---

<sup>1</sup> U ovom slučaju termin vidljivost ne odnosi se na to da li neko može da vidi da li se u kodu neke klase nalazi član (atribut ili metoda), već se odnosi na mogućnost pristupa tom članu klase kada koristimo objekat te klase.

```

class Radnik {
    private _ime: string;
    private _radnoVremeNedeljno: number;
    private _ukupnoOdradjenoSatiNedeljno: number;
    private _osnovica: number;
    private _koeficijentRada: number;

    public constructor(){
    }

    public povecajOdradjenoSati(sati: number): void { ... }
    public izracunajPlatu(): void { ... }
    public preostaloSatiNedeljno(): number { ... }
}

```

Ako bi pokušali direktno da menjamo vrednost atributa kao u primeru sa referencama, dobili bismo grešku od kompajlera jer ti atributi više nisu vidljivi. Za pristup private atributima obično se pišu public metode koje njima pristupaju, takozvani geteri i seteri. NAPOMENA: Po konvenici imenovanja u TypeScript private atributi bi trebalo da počnu sa donjom crtom “\_”. (Nije obavezno poštovati konvencije)

## 1.6. Sakrivanje informacija i enkapsulacija

Jedan od bitnih principa objektnog programiranja je princip sakrivanja informacija ([information hiding](#)), koji predstavlja sakrivanje i zaštitu implementacionih detalja jedne klase od drugih delova programa. Drugim rečima, objekti u programu treba da komuniciraju međusobno isključivo upotrebom dobro definisanih interfejsa (tj. public metoda). Neke od prednost koju nam donosi ovaj princip je: smanjenje potencijalnih grešaka koje bi napravili programeri prilikom neadekvatne upotrebe klase sa čijim implementacionom detaljima nisu upoznati; omogućava lakšu ponovnu upotrebu klase u drugim delovima programa; znatno smanjuje posao prilikom održavanja/promene implementacije neke klase.

U opštem slučaju, preporuka u objektnom programiranju je da ako nešto može da bude private u klasi to i treba da bude private u klasi.

Kao trivijalan primer možemo posmatrati implementaciju klase trougao. Trougao se sastoji od 3 stranice a, b, c, koje prave zatvorenu figuru čiji unutrašnji zbir uglova je 180°.

Trougao
+a : double +b : double +c : double

Slika 7. Trougao sa public atributima



Ukoliko imamo klasu kao na slici 8, sa public atributima, programer koji koristi tu klasu može da napiše sledeće:

```
let t: Trougao = new Trougao  
t.a = 12;  
t.b = 3;  
t.c = 4;
```

Gore navedeni kod je sintaksno ispravan, ali suštinski pogrešan jer programer koji je koristio klasu trougao nije bio dovoljno upoznat sa činjenicom da trougao ne može da ima stranice tih dimenzija. Odnosno dobili smo figuru sledećeg oblika:



Slika 8. Nepravilna figura

Takva figura bi prouzrokovala grešku u radu programa kada bi je koristili u našem programu kao trougao (npr. u jednom trenutku treba da se izračunaju uglovi, ili visina trougla i sl.).

Sakrivanje implementacionih detalja bi sprečilo da nam se dogode takve greške, tj. programer koji treba da napiše tu klasu bi napisao i metodu za postavljanje stranica trougla koja bi vršila sve bitne provere (da li stranice zapravo pripadaju trouglu i sl.) i tek onda izmenila vrednosti atributa. Čime bi izbegli grešku koja se desila u kodu iznad.

Trougao
-a : double -b : double -c : double
+postaviStranice() : boolean +getA() : double +setA() : void +getB() : double +setB() : void +getC() : double +setC() : void

Slika 9. Trougao sa private atributima

Kao što se vidi na slici 9. atributi u klasi se obično proglašavaju privatnim. Da bi se moglo pristupati tim atributima potrebno je kreirati public metode get i set za svaki pojedinačni atribut (popularni geteri i seteri). Na taj način se može kontrolisati mogućnost pisanja (samo set metoda) i mogućnost čitanja vrednosti atributa (samo get metoda). Konvencija pisanja naziva metoda je get + ime atributa ili set + ime atributa. U set metodi se često dodaje i validacija unetih podatak, tj. provera da li je nova vrednost koja je prosleđena ispravna.

U TypeScriptu, implementacija getera i setera se razlikuju od implementacije getera i setera u klasični OO jezicima tako što se geteri i seteri ne tretiraju kao obične metode:

- Za kreiranje getera i setera koriste se ključne reči **get** i **set** pre imena metode
- Metode getter i setera se ne pozivaju kao ostale metode (upotrebom zagrada ( ) ), već se koriste kao što bi koristili public atribut
- Pristupa im se isključivo sa dot (.) notacijom
- Po konvenciji imenovanja get i set metode trebalo bi da imaju isti naziv (bez donje crtice) kao i private atribut kome pristupaju

Primer:

```
class Trougao {
    private _a: number;
    private _b: number;
    private _c: number;

    public get a(): number { return this._a; }

    public set a(v: number) {
        //Ukoliko želimo da imamo validaciju mozemo
        if (v + this._b > this._c && v + this._c > this._b && this._b + this._c > v) {
            this._a = value;
        } else {
            console.log("Nije uneta stranica trougla");
        }
    }

    public get b(): number { return this._b; }
    public set b(value: number) { this._b = value; }
    public get c(): number { return this._c; }
    public set c(value: number) { this._c = value; }

}

let t: Trougao = new Trougao();
//Poziv setera
t.a = 2; t.b = 3; t.c = 4;

//Poziv gettera
console.log(t.a);
if(t.a > t.b){
    console.log("Pozvani geteri a i b");
}
```

## 1.7. this

Ključna reč „this“ u JavaScriptu se odnosi na objekat kome „this“ pripada. Upotreba ključne reči „this“ u JavaScriptu je malo složenija nego u klasičnim OO programskim jezicima, za više detalja možete pogledati [link](#). Ukratko:

- U metodi ili konstruktoru, „this“ se odnosi na objekat kome pripada metoda.
- U funkciji, „this“ se odnosi na globalni objekat.
- U eventu, „this“ se odnosi na dom element koji je primio event.
- Samostalno, „this“ se odnosi na globalni objekat.

U klasama „this“ koristimo kako bi referencirali instancu (objekat) koja izvršava kod, odnosno kako bi referencirali attribute ili metode tog objekta. Jedna od upotreba ključne reči „this“ je da reši problem dvosmislenosti (npr. parametar metode ima isti naziv kao atribut klase). Primer:

```
class Student {
    ime: string; //Atribut ime
    prosek: number; //Atribut prosek

    //Konstruktor ima paramtere ime i prosek
    constructor(ime: string, prosek: number){
        this.ime = ime;
        this.prosek = prosek;
        //ime = ime je dvosmisleno
        //this.ime se odnosi na atribut
        //ime se odnosi na parametar
    }
    ...
}
```

Kao što je već rečeno, ključna reč „this“ se koristi i u svim ostalim metodama klase koje referenciraju attribute te iste klase. Bilo je rečeno na šta se „this“ odnosi, u slučaju ako imamo metodu koja treba da doda funkciju koja će reagovati na neki event dom elementa (event listener) u toj funkciji nećemo moći da koristimo this kako bi referencirali attribute klase. Primer toga je metoda wireEvents u zadatku Calculator (Termin 7 – Zadatak 5):

```

class Calculator {
  private x : HTMLInputElement;
  private y : HTMLInputElement;
  private output : HTMLSpanElement;

  ...

  wireEvents() {
    document.getElementById('Add')
      .addEventListener('click', function(event){
        this.output.innerHTML = ...
      });
    document.getElementById('Subtract')
      .addEventListener('click', function(event){
        this.output.innerHTML = ...
      });
  }

  ...
}

```

U okviru `wireEvents` metode dodajemo event listnere na elemente sa id-jevima „Add“ i „Subtract“, tako da će „this“ u okviru tih funkcija odnositi na dom elemente „Add“ i „Subtract“ a ne na objekat klase. Što znači da linija koda koja pokušava da pristupi **output** atributu klase će rezultovati sa greškom (`this.output.inerHTML`). Postoji nekoliko mehanizama koji bi nam omogućili pristup atributima objekta u funkciji event listnera, ako za time ima potrebe, od kojih je korišćenje arrow funkcija najelegantniji:

```

wireEvents() {
  document.getElementById('Add')
    .addEventListener('click', event => {
      this.output.innerHTML = ...
    });
  document.getElementById('Subtract')
    .addEventListener('click', event => {
      this.output.innerHTML = ...
    });
}

```

Ovo funkcioniše zato što arrow funkcije nemaju svoj „this“, već u njima „this“ se odnosi na leksički kontekst u kome su iskorišćene. Drugim rečima, „this“ u arrow funkcijama se pronalazi kao što se pronalaze obične promenljive u funkcijama.

## 2. Nasleđivanje

Nasleđivanje se može definisati kao proces u kome jedna klasa preuzima (nasleđuje) osobine (atribute i metode) neke druge klase. Klasa koja se nasleđuje obično se zove natklasa (super class, ili roditeljska klasa) a klasa koja nasleđuje osobine se naziva potklasa (sub class, ili naslednik). Upotreba mehanizma nasleđivanja omogućava hijerarhijsko modelovanje klasa na osnovu njihovih odgovornosti i zaduženja.

Instanca objekta potklase će sadržati sve osobine koje sadrži i instanca natklase, pri čemu će objekat potklase imati dodatne osobine koje definiše potklasa. Zbog toga se u konstruktoru potklase poziva konstruktor natklase (kako bi se inicijalizovale nasleđene osobine).

U TypeScriptu nasleđivanje se postiže upotrebom ključne reči `extends` (pogledati kod u tabeli 1.). Postoji nekoliko tipova nasleđivanja:

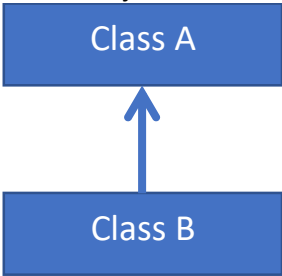
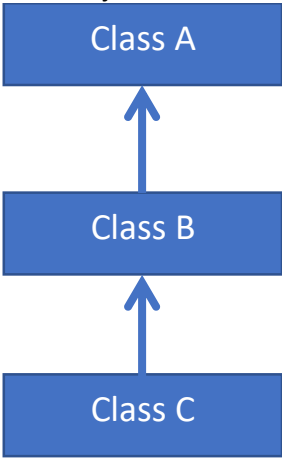
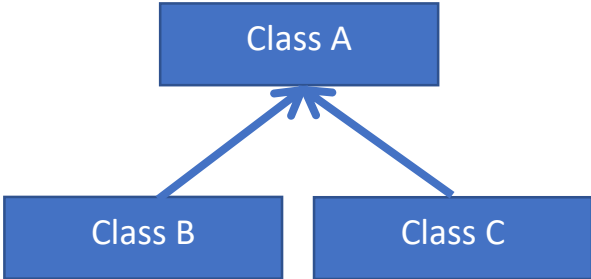
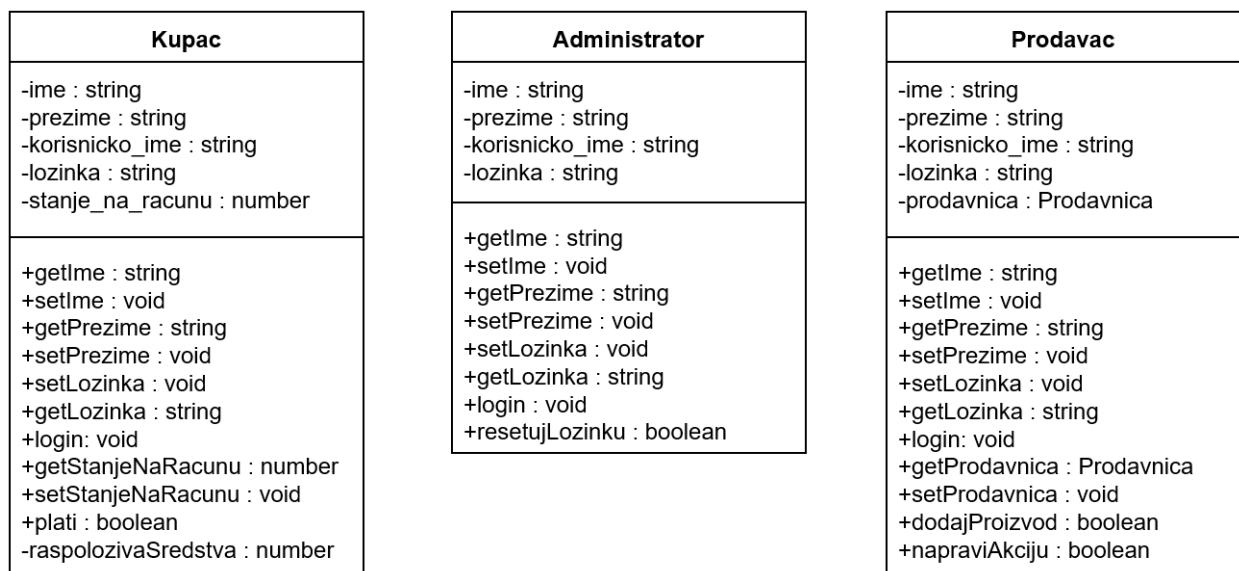
Tip nasleđivanja:	Kod:
Jednostruko nasleđivanje:  <pre>graph BT; B[Class B] --&gt; A[Class A]</pre>	<pre>class A { ... } class B extends A { ... }</pre>
Višesložno nasleđivanje:  <pre>graph BT; C[Class C] --&gt; B[Class B]; B --&gt; A[Class A]</pre>	<pre>class A { ... } class B extends A { ... } class C extends B { ... }</pre>
Hijerarhijsko nasleđivanje:  <pre>graph BT; B[Class B] --&gt; A[Class A]; C[Class C] --&gt; A</pre>	<pre>class A { ... } class B extends A { ... } class C extends A { ... }</pre>

Tabela 1. Tipovi nasleđivanja

Kao konkretan primer upotrebe nasleđivanja možemo da iskoristimo primer modelovanja korisnika aplikacije online prodavnice. Recimo da aplikacija treba da podržava tri tipa korisnika:

- Kupac – može da se loguje na aplikaciju, može da kupuje proizvode u aplikaciji, može da ostavlja komentare na proizvode i sl.
- Prodavac – može da se loguje na aplikaciju, može da dodaje proizvode za prodaju, može da odgovara na pitanja kupaca i sl.
- Administrator – može da se loguje na aplikaciju, zadužen za održavanje aplikacije.

Najjednostavniji način za modelovanje bi bio da se svaki od korisnika aplikacije predstavi kao posebna klasa (slika 10).

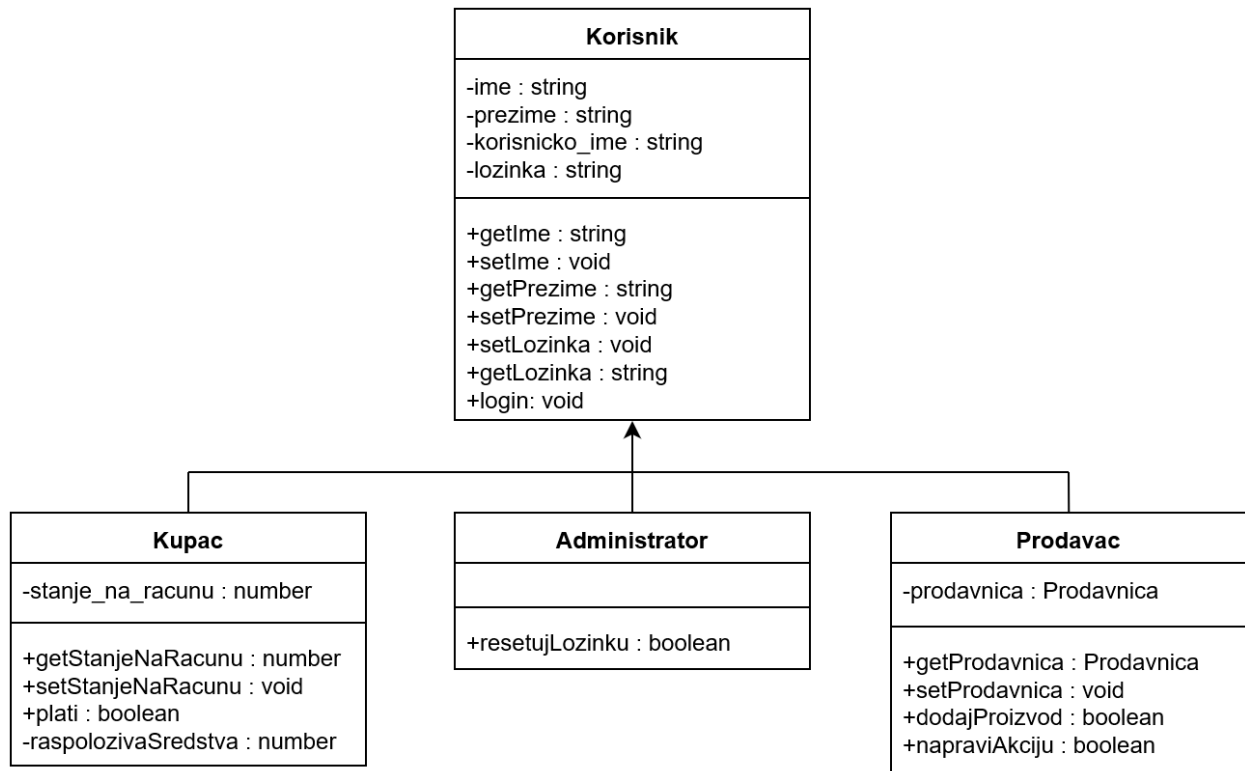


Slika 10. Modelovanje korisnika bez nasleđivanja

Kao što se vidi na slici, sve tri klase dele određene attribute (ime, prezime, korisnicko\_ime, lozinka) i metode (metode za rukovanje atributima, metoda za logovanje i sl.). Jedan od problema sa ovim načinom modelovanja je višestruka implementacija identičnih metoda na više mesta. Sve gore prikazane zajedničke metode rade identičan posao, i ne bi bilo loše da ih možemo implementirati na jednom mestu. To bi mogli da postignemo upotrebom nasleđivanja i sve zajedničke osobine da stavimo u natklasu. A ukoliko se kasnije mora održavati neka funkcionalnost (npr. promena načina logovanja korisnika), sa nasleđivanjem kod bi promenili na jednom mestu, a bez nasleđivanja isti kod bi se trebao menjati na tri različita mesta.

Još jedan od problema pristupa bez nasleđivanja je nedostatak polimorfizma (pogledati sekciju 2.1. Polimorfizam). Na primer, ukoliko je moguće ostavljati komentare na proizvode od strane korisnika, pri čemu je bitno pamtiti koji korisnik je ostavio komentar. Da bi to implementirali potrebno je da u proizvodu imamo metodu koja kao parametar prima korisnika. Sa pristupom bez nasleđivanja bi morali da u proizvodu implementiramo tri metode koje rade isti posao, ali primaju različit tip parametra (za svakog od korisnika po jedan). Ukoliko bi koristili nasleđivanje, mogli bismo da implementiramo samo jednu metodu koja kao parametar prima natklasu.

Na slici 11. je prikazan model sa korišćenjem nasleđivanja.



Slika 11. Modelovanje korisnika sa upotrebom nasleđivanja.

TypeScript kod za primer nasleđivanja između klasa **Korisnik** i **Kupac** je dat u tekstu ispod:

```

class Korisnik {
    //Ovi atributi bi mogli da budu protected
    //Radi jednostavnosti primera to je izostavljeno
    private _ime: string;
    private _prezime: string;
    private _korisnicko_ime: string;
    private _lozinka: string;

    constructor(i: string, p: string, ki: string, l: string) {
        this._ime = i;
        this._prezime = p;
        this._korisnicko_ime = ki;
        this._lozinka = l;
    }
}
  
```

```

    public get ime(): string { return this._ime; }
    public set ime(v: string) { this._ime = v; }
    public get prezime(): string { return this._prezime; }
    public set prezime(v: string) { this._prezime = v; }
    public get korisnicko_ime(): string { return this._korisnicko_ime; }
    public set korisnicko_ime(v: string) { this._korisnicko_ime = v; }
    public get lozinka(): string { return this._lozinka; }
    public set lozinka(v: string) { this._lozinka = v; }

    public login(): void {
        //LOGIKA LOGOVANJA
        //...
    }
}

class Kupac extends Korisnik {
    private _stanje_na_racunu: number;

    public constructor(i: string, p: string, ki: string, l: string, snr: number)
    {
        super(i, p, ki, l);
        this._stanje_na_racunu = snr;
    }

    public get stanje_na_racunu(): number {
        return this._stanje_na_racunu;
    }

    public set stanje_na_racunu(value: number) {
        this._stanje_na_racunu = value;
    }

    public plati(): boolean{
        //LOGIKA PLACANJA
        return true;
    }

    private raspolozivaSredstva(): number {
        //LOGIKA RACUNANJA RASPOLOZIVIH SREDSTAVA
        return 0;
    }
}

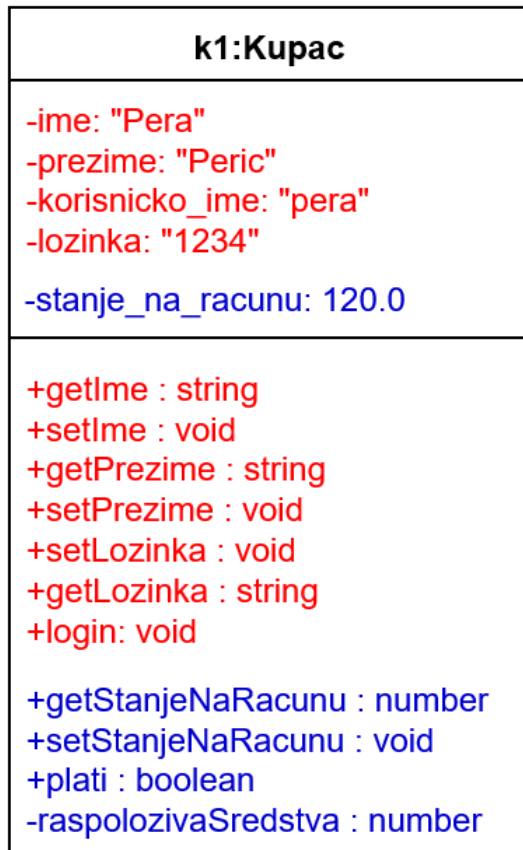
```

Obratiti pažnju na konstruktor klase Kupac. Taj konstruktor kao parametre treba da primi (nije obavezno) sve parametre koje prima roditeljski konstruktor i parametre koji će se iskoristiti za inicijalizaciju njegovih



atributa. U konstruktoru klase Kupac je pozvan konstruktor natklase (klase Korisnik), upotrebom ključne reči super, kako bi se inicijalizovali nasleđeni atributi. Instanca klase Kupac (objekat) će sadržati attribute i metode iz obe klase, primer:

```
let k1: Kupac = new Kupac("Pera", "Peric", "pera", "1234", 120.0)
```



Slika 12. Objekat klase Kupac

## 2.1. Polimorfizam

Po jednoj od definicija, polimorfizam (u programskim jezicima) je obezbeđivanje jedinstvenog interfejsa entitetima različitih tipova. Polimorfizam često se koristi u programskim jezicima i znatno olakšava implementaciju funkcionalnosti koje su zajedničke za više klasa. Najčešća upotreba polimorfizma u OO jezicima je upotreba referenci natklase kako bi se referencirao objekat potklase (kao u primeru za dodavanje komentara kod korisnika online prodavnice). Primer:

```

let k:Korisnik = new Kupac("Pera", "Peric", "pera", "1234", 120

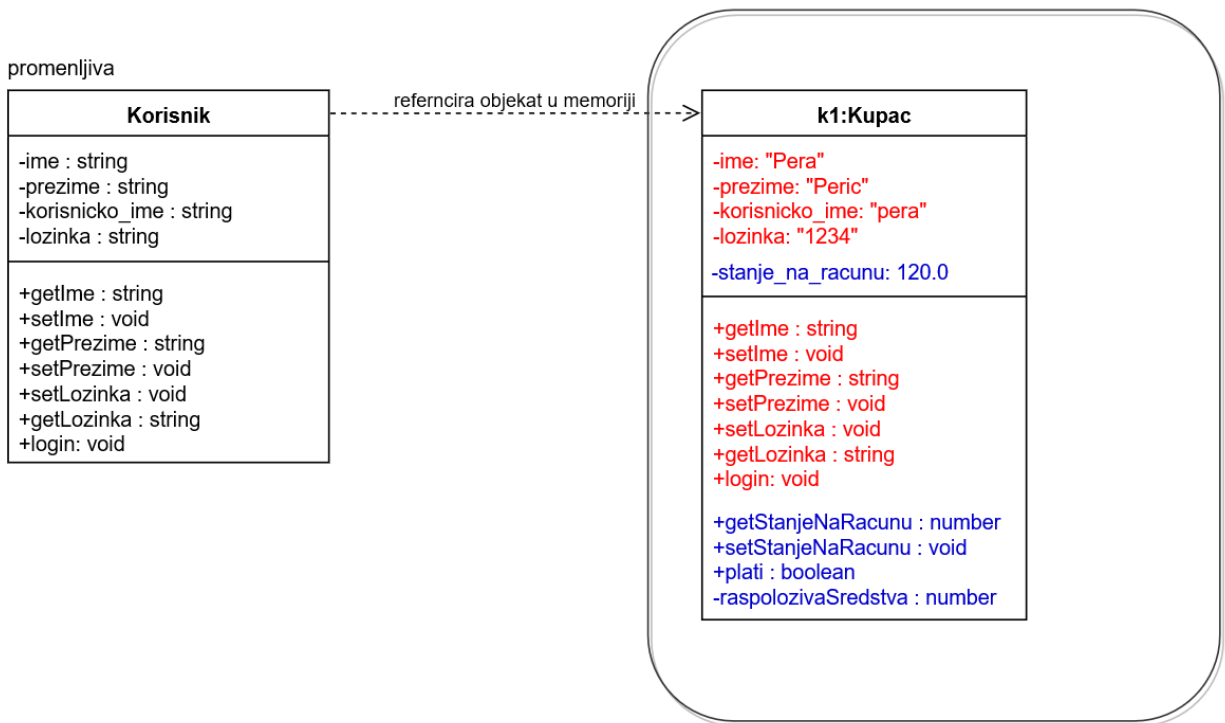
// ili

function fja(k: Korisnik): void { ... };

let kp: Kupac = new Kupac("a", "a", "a", "a", 1);
fja(kp);

```

Pošto klasa Kupac nasleđuje klasu Korisnik, objekat klase Kupac će sadržati sve osobine koje bi sadržao i objekat klase Korisnik (slika 12. osobine obeležene crvenom bojom). Odnosno, svaki Kupac je Korisnik, ali svaki Korisnik nije Kupac. Tako da je dodela objekta klase Kupac promenljivoj tipa Korisnik moguća, a u suprotnom smeru nije dozvoljeno zato što objekat nadklase ne bi sadržao sve osobine podklase (npr. osobine ofarbane plavom bojom na slici 12.).



Slika 13. Referenciranje objekta u memoriji

Pošto su promenljive zapravo reference na objekat, korisnik koji koristi referencu tipa nadklase će moći da pristupa osobinama samo za koje ta referenca zna (slika 13.). Iako taj objekat u sebi sadrži više podataka (atributa i/ili metoda) korisnik će moći samo da koristi ono što je poznato tipu te promenljive, u konkretnom primeru samo osobine klase Korisnik.

## 2.2. Redefinisanje metoda (method overriding)

Redefinisanje metoda (method overriding) predstavlja mogućnost da se u potklasi promeni ponašanje (odnosno reimplementira) metoda natklase. Redefinisanje se postiže tako što se u potklasi implementira metoda koja ima isto ime, parametre i povratnu vrednost kao i metoda u natklasi. Redefinisanjem metodu u potklasi možemo da proširimo (tj. ukoliko smo pozvali roditeljsku metodu upotrebom ključne reči **super**) ili da promenimo (tj. implementiramo metodu kao što je to rađeno i do sada). Primer:

```
class Animal {
  name: string;
  constructor(theName: string) { this.name = theName; }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Horse extends Animal {
  constructor(name: string) { super(name); }

  //Method overriding
  move(distanceInMeters: number = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let tom: Animal = new Horse("Tommy the Palomino");
tom.move(34);
//Ispis u konzoli je:
//Galloping...
//Tommy the Palomino moved 34m.
```

## 2.3. Interfejsi

U mnogim OO jezicima klase podržavaju samo jednostruko nasleđivanje, tj. klase mogu da naslede samo jednu klasu. To može da bude veoma ograničavajuće, posebno kada određena klasa naslednica (i sve njene naslednice) kao i druge klase u hijerarhiji klasa treba da imaju neko zajedničko ponašanje.

U OO jezicima interfejs je kolekcija metoda koji ukazuju da klasa poseduje neko ponašanje pored onoga koje nasleđuje od natklase. Interfejs ne sadrži implementaciju metoda, već samo naznačava da klasa koja implementira interfejs mora da implementira i metode koje su navedene u interfejsu.

Neke od razlika između interfejsa i klasa su:

- Interfejs se ne može instancirati

- Interfejs ne sadrži konstruktore
- Sve metode u interfejsu su apstraktne (sadrže samo definiciju metoda)
- Interfejs ne sadrži polja tj. attribute – (Interfejsi u TypeScript ovo dozvoljavaju)
- Klase ne nasleđuju interfejse, već ih implementiraju upotrebom ključne reči implements
- Klasa može da implementira više interfejasa
- Interfejs može da nasledi više drugih interfejasa upotrebom ključne reči extends

Primer upotrebe interfejasa:

```
interface Animal {
    talk: () => void;
    noOfLegs: () => number;
}

class Dog implements Animal {

    public talk(): void {
        console.log("Av av av av!");
    }

    public travel(): void {
        console.log("Dog travels");
    }

    public noOfLegs(): number {
        return 4;
    }
}
```

Preporuke kada koristiti interfejs a kada klasu:

- Klase se obično koriste kada objekat te klase može samostalno da postoji u sistemu
- Interfejse obično koristimo kada objekat tog tipa ne treba samostalno da postoji u sistemu, već želimo da taj objekat (odnosno neka klasa) ima implementaciju određenih osobina
- Ukoliko je potrebno u roditelju implementirati metodu koja je zajednička za sve naslednike, a objekat tog tipa ne treba samostalno da postoji u sistemu, za to se obično koriste [apstraktne klase](#).
  - Apstraktne klase se ne mogu instancirati kao ni interfejsi, ali za razliku od interfejasa mogu da sadrže attribute i implementaciju metoda, ali mogu da sadrži i apstraktne metode kao interfejsi
  - Primer gde bi koristili apstraktnu klasu je primer klase Korisnik online prodavnice, ne bi trebalo da postoji objekat tipa Korisnik u sistemu (već objekti Kupac, Prodavac ili Administrator) ali je potrebno implementirati određene zajedničke metode poput login, getera i setera i sl.