

Autori: © Divna Krpan, Saša Mladenović, Goran Zaharija

# OOP Vježbe 04

## Svojstva i polja klase

### Bilješke

#### Svojstva i polja

Klase su dio ECMAScript 2015 standarda, što znači da nisu podržane u starijim web preglednicima. Međutim, prethodni način pisanja objektno orijentiranih programa u JavaScript-u je bio nepregledan u usporedbi sa C#-om.

**Polja** (eng. fields) predstavljaju attribute koje će objekti imati prilikom instanciranja. Svi objekti koji pripadaju istoj klasi imaju ista polja, ali svaki objekt ima svoju vlastitu vrijednost koja je spremljena u to polje. Ponekad se u literaturi za podatkovnu strukturu **Array** koristi prijevod "polje", ali se u ovim materijalima za **Array** koristi prijevod "niz". U kontekstu klase i objekta, pojam "polje" odnosi se na jedan atribut klase, koji može biti predstavljen bilo kojim tipom podataka (uključujući i niz). Na primjer, ako definiramo klasu *Student*, koja ima polja: *ime*, *prezime* i *ispiti*, možemo stvoriti dva različita objekta:

```
class Student {  
  constructor(imeS, prezimeS) {  
    this.ime = imeS;  
    this.prezime = prezimeS;  
    this.ispiti = [];  
  }  
}  
let a = new Student("Ante", "Antić");  
let b = new Student("Iva", "Ivić");
```

Objekti *a* i *b* imaju različite vrijednosti polja. Polja su zapravo varijable koje sadrže podatke.

U C#-smo razlikovali razine pristupa: **private** i **public**. Razina **private** omogućavala je pristup poljima i metodama samo unutar klase u kojima su definirani, dok su **public** polja i metode bili dostupni van klase. Međutim, ako želimo zaštititi sadržaj polja ili sačuvati stanje objekta od

neovlaštenog pristupa, ali ipak želimo omogućiti neki način pristupa tim podacima, onda definiramo **svojstva** (eng. properties). Svojstva ograničavaju pristup sadržaju polja te moraju biti *public*. U C#-u ih definiramo pomoću **get** i **set**.

U JavaScript-u nemamo ključne riječi *private* ili *public* kojima bi mogli odrediti razinu pristupa, no možemo definirati svojstva pomoću *get* i *set*. Umjesto ključne riječi *private* privatne varijable ćemo implementirati pomoću donje crte, kao što možemo vidjeti na primjeru imena:

```
this._ime = "vrijednost";
```

Donja crta će nam po dogovoru označavati privatnu varijablu, premda je i dalje moguće pristupiti varijabli `_ime` iz drugih dijelova odnosno klasa. JavaScript u tome nije toliko “strog” kao C#, ali nam ipak omogućuje pisanje u objektno orijentiranom stilu. Pisanje programa u objektno orijentiranoj paradigmi je odgovornost programera.

Nadalje, definiramo **pristupne metode** za **dohvaćanje** (eng. *get*) i **postavljanje** (eng. *set*) vrijednosti varijable `_ime`.

Pristupna metoda za dohvaćanje sadržaja varijable `_ime`:

```
get ime(){
    return this._ime;
}
```

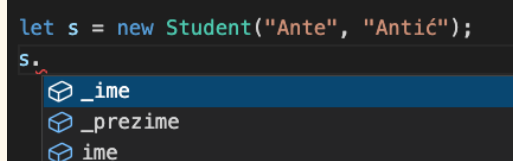
Pristupna metoda za postavljanje sadržaja varijable `_ime`:

```
set ime(vrijednost) {
    this._ime = vrijednost;
}
```

Ako želimo postaviti vrijednost varijable u ostatku programa, jednostavno nastavimo koristiti svojstvo tako da na prvi pogled nema razlike niti se ostatak programa promijenio:

```
// spremanje
s.ime = "Pero";
// čitanje
console.log(s.ime);
```

Na sljedećoj slici možemo vidjeti kako su varijable s donjom crtom i dalje vidljive:



```
let s = new Student("Ante", "Antić");
s.
  _ime
  _prezime
  ime
```

Gornji primjer za postavljanje imena studenta trenutno nema svrhu jer ne pruža nikakvu dodatnu funkcionalnost osim spremanja i čitanja vrijednosti, a to smo već imali korištenjem samog polja. Kako bi definiranje svojstva imalo smisla, potrebno je odrediti što će to svojstvo raditi, a da se razlikuje od običnog polja ili varijable.

Definirajmo svojstvo koje će u slučaju kad pokušamo spremiti prazno ime upisati vrijednost "nepoznato":

```
set ime(vrijednost) {
  if (vrijednost == "")
    this._ime = "nepoznato";
  else
    this._ime = vrijednost;
}
```

Prema tome, ako probamo sljedeće naredbe:

```
// spremanje
s.ime = "";
// čitanje
console.log(s.ime); //ispisuje "nepoznato"
```

primijetiti ćemo kako se u polje `_ime` ne može spremiti prazan string.

Uzmimo primjer klase *Radnik*:

```
class Radnik {
  constructor(imer) {
    this.ime = imer;
    this.racun = 100;
  }

  isplata(i) {
    console.log("Isplata: " + i);
    this.racun -= i;
  }
}
```

Klasa ima polja `ime` i `racun`. Metoda za isplatu umanjuje stanje računa za upisani iznos. Ako pokušamo izvršiti sljedeće naredbe:

```
let r = new Radnik("Pero");
console.log(r);

r.isplata(200);
```

```
console.log("Stanje: " + r.racun);

r.isplata(50);
console.log("Stanje: " + r.racun);
```

Dobit ćemo sljedeći ispis:

```
► Radnik {ime: 'Pero', racun: 100}
Isplata: 200
Stanje: -100
Isplata: 50
Stanje: -150
```

Ako želimo spriječiti negativno stanje računa, potrebno je ograničiti pristup polju *racun*, odnosno napisati svojstvo koje će nam to omogućiti. Ako se želi preuzeti više nego što ima na računu, treba ispisati poruku koliko se može preuzeti i stanje postaviti na 0.

Obzirom da smo već započeli program u kojem je na više mjesta korištena varijabla *racun*, najjednostavnije je u ovom trenutku samo promijeniti naziv varijable u konstruktoru te definirati pristupne metode za dohvaćanje i spremanje stanja računa:

```
class Radnik {
  constructor(imer) {
    this.ime = imer;
    this._racun = 100; // novi naziv!
  }

  isplata(i) {

    console.log("Isplata: " + i);
    this.racun -= i;
  }

  // svojstvo racun:
  get racun() {
    return this._racun;
  }

  set racun(vrijednost) {
    this._racun = vrijednost;
  }
}
```

```
}
```

Nakon ove prve intervencije, izmjene naziva varijable i definiranjem svojstva, ostatak programa ostaje isti i program se ponaša isto. No, cilj je spriječiti negativno stanje računa, a to možemo napraviti kod postavljanja vrijednosti, odnosno u *set* dijelu na primjer ovako:

```
set racun(vrijednost) {
  if (vrijednost < 0) {
    console.log("Stanje ne može biti negativno!");
    console.log("S računa možete preuzeti: " + this._racun);
    this._racun = 0;
  }
  else
    this._racun = vrijednost;
}
```

Nakon ovih izmjena, program će ispisati sljedeće:

```
► Radnik {ime: 'Pero', _racun: 100}
Isplata: 200
Stanje ne može biti negativno!
S računa možete preuzeti: 100
Stanje: 0
Isplata: 50
Stanje ne može biti negativno!
S računa možete preuzeti: 0
Stanje: 0
```

Dakle, u trenutku kad smo pokušali preuzeti 200 kuna s računa, metoda *isplata()* je pokušala oduzeti 200 i spremiti -100 kao novo stanje. Međutim, *set* metoda je spriječila postavljanje negativne vrijednosti. Ispisano je trenutno stanje (koliko ima na raspolaganju) i postavljeno stanje na 0. U prvom pokušaju isplate smo već trebali “ići u minus” na računu, ali stanje je postalo 0. Pri svakom idućem pokušaju će stanje i dalje ostati 0. Na ovaj način smo omogućili implementaciju **enkapsulacije**.

Ako u C#-u ne navedemo je li neka varijabla unutar klase *public* ili *private*, onda je zadano da je *private*. Međutim, u JavaScript-u zadana razina pristupa za sve varijable i metode je *public*.

Prefiks naziva polja u obliku donje crte je **konvencija** ili dogovor kako bi takve varijable tretirali kao privatne iako su zapravo i dalje javno dostupne.

Novi prijedlog je korištenje prefiksa # umjesto donje crte. Kako bi mogli koristiti tu novu opciju, potrebno je privatnu varijablu **#racun** dodatno deklarirati unutar klase, ali van konstruktora, kao što možemo vidjeti u sljedećem primjeru:

```
class Radnik {
  #racun;
  constructor(ime) {
    this.ime = ime;
    this.#racun = 100;
  }

  isplata(i) {

    console.log("Isplata: " + i);
    this.racun -= i;
  }

  // svojstvo racun:
  get racun() {
    return this.#racun;
  }

  set racun(vrijednost) {
    if (vrijednost < 0) {
      console.log("Stanje ne može biti negativno!");
      console.log("S računa možete preuzeti: " + this.#racun);
      this.#racun = 0;
    }
    else
      this.#racun = vrijednost;
  }
}

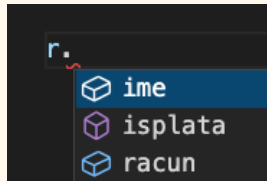
let r = new Radnik("Pero");
console.log(r);

r.isplata(200);
console.log("Stanje: " + r.racun);

r.isplata(50);
```

```
console.log("Stanje: " + r.racun);
```

Ako koristimo # kao prefiks, onda privatna varijabla neće biti vidljiva u ostatku programa:



Navedeni način pisanja privatnih polja s prefiksom # spada u relativno novi prijedlog čija objava se očekuje u 2022. godini. Iako možemo vidjeti da novi način radi na nekim operacijskim sustavima i preglednicima, obzirom da je to još relativno novo, ipak ćemo koristiti konvenciju s donjom crtom kako ne bi imali problema kod starijih web preglednika. Konvencija pisanja s # oznakom pokriva i privatne metode. Dodatno je korištenje donje crte jednostavnije jer ne moramo posebno deklarirati varijablu van konstruktora.

✖ Uncaught SyntaxError: Private field '#racun' must be declared in an enclosing class

U literaturi možete pronaći više različitih načina na koje se može implementirati koncept svojstva. Dovoljno je poznavati ili koristiti u projektu prvi način s donjom crtom.

### *Primjena svojstva u izradi igre*

Na prošlim vježbama smo naučili pomicati likove po canvasu u 4 smjera izmjenama  $x$  i  $y$  koordinata. Problem kojeg ste trebali riješiti samostalno je bio kako omogućiti crtanje više likova odjednom. Razlog problema je bilo “osvježavanje” canvasa prije crtanja svakog lika unutar metode `draw()`. Jedan od prijedloga rješenja je definiranje nove metode koja će crtati sve likove odjednom i osvježavati sliku prije svakog crtanja:

```
drawAll.sprites) {
  this.context.fillStyle = "#f2f2f2";
  this.context.fillRect(0, 0, 640, 480);
  for (let i = 0; i < sprites.length; i++) {
    const s = sprites[i];
    this.draw(s);
  }
}
```

Metoda `drawAll()` prima niz `sprites` koji predstavlja niz likova u igri. Očistit će područje crtanja kao i prošli put i nacrtati novo stanje svih likova. Pri tome smo opet iskoristili već postojeću metodu `draw()`.

Nadalje, obzirom da će očitito biti potrebno pratiti stanje likova na neki način, definirajmo klasu **Game** koja će omogućiti da na jednom mjestu vodimo računa o likovima u igri:

```
class Game {
  constructor() {
    //svi likovi koji se nalaze u igri
    this.sprites = [];
  }

  addSprite(s) {
    this.sprites.push(s);
  }
}
```

Unutar klase **Game** definiran je popis likova. Kako bi nam bilo lakše zapamtiti način dodavanja u niz u JavaScript-u, napravljena je dodatna metoda *addSprite()*.

U glavnom dijelu programa, deklarirali smo novu klasu kao konstantu (**const**) kako bi je mogli koristiti u cijelom programu i imali pristup likovima po potrebi:

```
const GAME = new Game();
```

U funkciji *setup()* dodani su novi likovi mačke i psa koji su dodatno spremljeni u popis svih likova igre.

```
function setup() {

  // dodavanje likova
  cat = new Sprite(slike.cat, 0, 0, 50, 50);
  dog = new Sprite(slike.dog, 0, 60, 50, 50);

  GAME.addSprite(cat);
  GAME.addSprite(dog);
}
```

U petlji igre pozivamo objekt **DISPLAY** kojeg smo nazvali velikim početnim slovima kako bi ga razlikovali od ostalih varijabli. Samo je jedno područje na kojem crtamo i kojim upravlja **DISPLAY** objekt.

```
// petlja igre - za crtanje
function gameLoop(time) {
  // console.log(time);
```



```

DISPLAY.drawAll(GAME.sprites);

window.requestAnimationFrame(gameLoop);
}

```

Također, samo je jedna aktivna igra, odnosno imat ćemo jedan objekt **GAME** kojeg ćemo koristiti u ovom primjeru.

U gotovom primjeru kojeg ste mogli preuzeti možemo primijetiti jedan novi problem:



Naime, ako kliknemo na *Start* više puta, stvorit će se više likova. Ovo nije problem kao na prošlim vježbama gdje su nam ostajali tragovi lika već stvarno postoje likovi (objekti) u igri koji su ostali od prethodnog pokretanja u popisu svih likova (*GAME.sprites*). To je situacija koju ćete imati kod pokretanja vašeg projekta gdje će biti potrebno odlučiti treba li počistiti postojeće stanje nakon završetka igre ili se prelazi na novu *razinu* (eng. level) gdje će svi ili neki od likova nastaviti dalje.

Obzirom na trenutni primjer gdje možemo upravljati samo s dva lika, potrebno je najprije počistiti trenutno stanje definiranjem dodatne metode *clearSprites()* koja će pobrisati sve likove u igri prije stvaranja novih. Dakle, pozvat ćemo tu metodu kod postavljanja likova: *setup()*. Spomenuti primjer je pojednostavljeni prikaz buduće strukture okvira za izradu projekta gdje se objekt *GAME* brine o stanju igre.

Ako dovoljno puta kliknemo na naredbe za pomicanje (trenutno botune), primijetit ćemo kako likovi izlaze van prostora predviđenog za crtanje. Prošli put smo vidjeli kako možemo definirati metode za pomicanje na više različitih načina te kako možemo imati više takvih metoda unutar našeg programa. Kako to promijeniti?

Možemo promijeniti metode za crtanje dodavanjem naredbe koja će odlučiti smije li se lik pomaknuti u nekom smjeru ili ne. Problem je što imamo 4 metode za 4 smjera koje smo koristili za petu metodu. Dakle, imamo najmanje dvije metode za pomicanje po osi x (petu metodu *move()* je bilo moguće implementirati i bez pozivanja prethodnih metoda čime bi dobili još

jednu metodu koja izravno pristupa koordinatama lika). U ovom programu bi mogli očekivati još neku metodu za pomicanje, npr. skok na lijevu ili desnu stranu. U tom slučaju bi opet morali dodati uvjet kojim spriječavamo pomicanje lika van granica. Tako zapravo na više mjesta ograničavamo pristup ili mogućnost promjene koordinata x i y

Dakle, iskoristit ćemo znanje o svojstvima kako bi spriječili postojeće i buduće metode u pomicanju lika van granica. Vodite računa kako su dimenzije trenutnog canvasa fiksne: 640 x 480 piksela.

➤ Zadatak 1.

- Napišite svojstvo x koje će ograničiti kretanje likova po osi x.
- Najprije riješimo problem lijeve strane. Likove možemo pomaknuti lijevo pomoću konzole.
- Riješite problem desne strane. Pazite, lik ne smije nestati.

Problem kod rješavanja ovog zadatka stvara nam i način pokretanja lika. Bilo bi dosta jednostavnije kad bi se lik mogao pomicati pritiskom na tipku na tipkovnici umjesto pisanja naredbi u konzoli ili klikanja po botunima. Za tu namjenu moramo upoznati pojam događaja.

### *Događaji*

Općenito, **događaji** (eng. events) su akcije koje može uzrokovati korisnik tijekom interakcije sa stranicom, npr. klik ili pomak miša, pritisnuta tipka na tipkovnici i sl. Događajima upravljamo registracijom odgovarajuće funkcije. To možemo obaviti na više načina. Primjer za događaj klik možete vidjeti u postojećem programu (pojednostavljeni okvir u kojem pomičemo likove).

Naredba kojom dohvaćamo HTML element *button*:

```
let btnCat = document.getElementById("btnCat");
```

1. način za dodavanje funkcije za upravljanje događajem:

```
btnCat.onclick = pomicanjeMacke;
```

Na taj način možemo dodati samo jednu funkciju za upravljanje klikom.

2. način:

```
btnCat.addEventListener("click", pomicanjeMacke);
```

Primjenom drugog načina možemo dodati više od jedne funkcije za upravljanje događajem.

Naučit ćemo kako dodati mogućnost upravljanja tipkama na tipkovnici na sličan način kao što smo to vidjeli na primjeru klika. Kako bi to mogli napraviti potrebno je:

- Otkriti je li tipka na tipkovnici pritisnuta
- Koja tipka je pritisnuta
- Kako reagirati na tipku

Kada god se dogodi neki događaj, javascript sprema sve podatke vezane za taj događaj u odgovarajući objekt (npr. gdje je bila strelica miša, koja tipka je bila pritisnuta, na koji element je kliknuto, ...). Taj objekt se proslijeđuje *callback* funkciji. U donjem primjeru je to parametar kojeg smo nazvali *event*.

```
function pomicanjeMacke(event) {
    console.log(event);
    cat.move(50);
}

function pomicanjePsa() {
    dog.move(50);
}
```

Naredba za ispis u konzoli će nam dati informaciju o kojem događaju se radi te sve ostalo što je spremljeno u objektu. U gornjem primjeru *pomicanjeMacke()* radi se o **PointerEvent** vrsti događaja. U primjeru *pomicanjePsa()* nam nije bio potreban događaj pa ga nismo niti naveli kao parametar. U C#-u nije dozvoljeno takvo izostavljanje parametara čak niti u situaciji kad nam ne trebaju.

Objekt **window** predstavlja prozor web preglednika te sadrži razna svojstva i metode koje nam omogućuju i određenu razinu kontrole nad web preglednikom (naravno, u skladu sa sigurnosnim postavkama). Npr. možemo doznati dimenzije ili položaj prozora. Kod našeg problema detektiranja događaja pritiskanja tipke, iskoristit ćemo objekt **window** za osluškivanje događaja te mu pridružiti funkciju koja će reagirati na događaj:

```
window.addEventListener("keydown", provjeriZnak);
function provjeriZnak(event) {
    console.log(event);
}
```

Kod pritiska tipke na tipkovnici razlikujemo tri vrste događaja:

- *keydown* → tipka pritisnuta prema dolje
- *keyup* → otpuštanje tipke
- *keypress* → oba događaja zajedno (dolje i gore)

Nama su zanimljivi *keydown* i *keyup* koji nam omogućuju i praćenje situacije kad korisnik stalno drži tipku pritisnutu. Na primjer, kod skakanja u desno, možemo držati tipku za kretanje u

desno i tipku kojom skačemo prema gore. Naš program mora biti u mogućnosti pratiti koje su tipke pritisnute prema dolje u isto vrijeme.

Kod ispisa objekta *event* na konzolu nakon pritisnute tipke “*a*” na tipkovnici možemo vidjeti svojstva:

- `event.code: "KeyA"`
- `event.key: "a"`
- `event.keyCode: 65`

Pomoću ta tri svojstva možemo identificirati tipku. Jedino *event.key* omogućuje razlikovanje velikog i malog slova *a*. Ako ne moramo razlikovati velika i mala slova, onda ćemo koristiti jedan od preostala dva načina.

➤ Zadatak 2.

- Dodajte u postojeći program mogućnost kretanja mačke na pritisak tipke s tipkovnice:
  - Ako pritisnemo tipku d, neka se mačka kreće desno
  - Ako pritisnemo tipku a, neka se mačka kreće lijevo
- Dodatno, napravite i preostala dva smjera:
  - w: gore
  - s: dolje

–