Autori: © Divna Krpan, Saša Mladenović, Goran Zaharija

OOP Vježbe 03
Objekti, klase i metode

—

Bilješke

Što ćemo naučiti?

- Pisanje klasa u JavaScript-u
- Metode i funkcije
- Stvaranje objekata klase
- Canvas i crtanje
- Petlja igre

Funkcije

Funkcija u JavaScriptu je kao *potprogram* ili *blok koda* koji se sastoji od niza naredbi. Funkciju definiramo ključnom riječi ***function*** iza koje slijedi naziv funkcije i popis parametara u okruglim zagradama (ako ih ima) te tijelo funkcije omeđeno vitičastim zagradama:

```
function mojaFunkcija(){
 // naredbe
}
```

Ako funkcija vraća vrijednost, onda mora sadržavati naredbu **return**. Primijetimo da nije bilo potrebno definirati tip povratne vrijednosti u zaglavlju funkcije, kao što smo to morali raditi kod definiranja metoda u C#-u.

Također, u JavaScript-u postoji više načina na koje možemo definirati funkcije. U sljedećem primjeru možemo vidjeti funkciju koja je definirana i spremljena u varijablu pomoću *anonimne funkcije* (eng. anonymous function expression) gdje se ne koristi naziv funkcije.

```
let mojaFunkcija = function() {
 // naredbe
}
```

Sigurno ćete upoznati još načina za definiranje funkcija, ali ovdje ćemo najčešće koristiti prvi spomenuti način.

Istaknimo:

- Funkcija će se izvršiti kad je pozvana u nekom dijelu programa.
- Funkcija se poziva pisanjem naziva funkcije, okruglih zagrada i parametara.
- Naziv može sadržavati slova, znamenke (ne smije početi znamenkom), znak dolara i donju crtu.
- Parametri ili argumenti funkcije se navode unutar okruglih zagrada koje slijede odmah nakon naziva funkcije i za njih također nije potrebno navoditi tip.
- Čim se izvrši naredba **return**, prekida se izvođenje funkcije te se vraća povratna vrijednost u dio programa koji je pozvao funkciju.

Važno: Ako je funkcija definirana tako da očekuje parametre, a kod poziva ih ne navedemo, to neće biti sintaksna pogreška kao u C#-u već će parametri biti vrijednosti ***undefined***.

Provjerimo na primjeru zbrajanja dva broja:

```
function zbroj(a, b) {
 return a + b;
}

console.log(zbroj(2, 3)); //Ispis: 5
console.log(zbroj()); // Ispis: NaN
```

U prvom pozivu funkcije *zbroj()* poslane su dvije vrijednosti: 2 i 3, a funkcija ispisuje očekivani rezultat. Međutim, kod drugog poziva izostavljeni su parametri. Obzirom da je u tom slučaju vrijednost od a i b jednaka ***undefined***, nije bilo moguće izračunati njihov zbroj te je vraćeno **NaN**. To je skraćenica od "*Not-a-Number*", a **NaN** je broj koji nije ispravan. Ipak pripada brojevima jer, ako pokušate ispisati tip od NaN pomoću naredbe ***typeof***, dobit ćete rezultat *number*.

Inače, postoji nekoliko slučajeva kad se dobije NaN:

- Broj se ne može parsirati iz teksta: (npr. `parseInt("tekst")`).
- Rezultat matematičke funkcije koji nije realan broj (npr. `Math.sqrt(-1)`).
- Jedan operand u aritmetičkom izrazu je NaN (npr. `5 * NaN`).
- Neodređena vrijednost (npr. `0 * Infinity, undefined + undefined`).
- Bilo koja aritmetička operacija koja uključuje string, a nije zbrajanje (npr. `"abc" / 3`).

Metode

Metode u su akcije koje izvode objekti ili akcije koje su definirane za neki objekt, a u JavaScript-u metoda je zapravo svojstvo koje sadrži definiciju funkcije (što trenutno zvuči malo čudno).

Uvođenjem standarda ECMAScript 2015, omogućeno je kraće pisanje metoda u obliku:

```
let poruka = {
 pozdravi(ime){
   return "Dobar dan " + ime;
 }
}

console.log(poruka.pozdravi("Pero"));

// Ispis: Dobar dan Pero
```

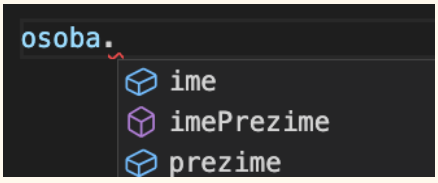
Postoji više načina na koje se metode mogu pisati, ali ovdje je odabran upravo taj kraći način pisanja. U gornjem primjeru, objekt *poruka* sadrži metodu *pozdravi()* koja prima *ime* osobe koju treba pozdraviti. Metoda također ima pristup sadržaju samog objekta (poljima), pomoću ključne riječi **this** kako smo navikli u C#-u:

```
let osoba = {
 ime: "Pero",
 prezime: "Perić",

 imePrezime() {
   return this.ime + " " + this.prezime;
 }
};

console.log(osoba.imePrezime());
// Ispis: Pero Perić
```

Dok tipkamo u VS Code, otvara se pomoćni prozor te po ikonicama možemo razlikovati što točno pozivamo:



Place "kockice" predstavljaju *svojstva* (eng. properties), a ljubičaste predstavljaju metode. Nije svejedno jer metode imaju okrugle zagrade (čak i kad nemaju parametre), a ako izostavimo okrugle zagrade kad se radi o metodi, nećemo dobiti upozorenje na sintaksnu pogrešku već se dobije ispis definicije metode:

```
console.log(osoba.imePrezime);
```

Ispis:

```
f imePrezime() {
    return this.ime + " " + this.prezime;
  }
```

Dakle, moramo biti pažljivi! Na isti način pozivamo i ugrađene metode:

```
let tekst = "Hello world!";
let veliki = tekst.toUpperCase();

console.log(veliki);
// Ispis: HELLO WORLD!
```

Opcionalni parametri

Prilikom deklariranja metoda, možemo definirati jedan ili više **opcionalnih parametara** na način da se odmah kod deklaracije parametara navedu i vrijednosti. Tako imamo osiguranu vrijednost parametara ako se u samom pozivu ne navedu. Obzirom da nemamo provjeru tipa, u JavaScript-u moramo još više paziti na poredak parametara. Opcionalne parametra bi trebali navesti na kraju jer kod pozivanja metode ne možemo preskakati parametre već ih metoda preuzima redom.

Vrijednosni i referentni tipovi

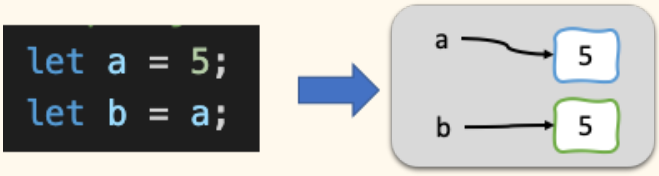
Važno je znati kako se ponašaju vrijednosni i referentni tipovi podataka u programskom jeziku u kojem radimo. Najčešće nećemo dobiti nikakvo upozorenje da smo napravili pogrešku već se program neće ponašati onako kako očekujemo. Ponekad to možemo otkriti jer program ne ispisuje očekivani rezultat, ali čim se radi o nešto većim programima (kao što ćemo raditi u projektima), onda može biti jako teško otkriti što se točno dogodilo.

Vrijednosni tipovi podataka su *jednostavni* tipovi podataka kao što su: number, boolean, string te null i undefined.

Referentni tipovi podataka su objekti (posebno "obični" objekti koje smo do sad koristili, nizovi i funkcije, ...).

Vrijednosni tipovi podataka

Deklarirat ćemo dvije varijable u koje ćemo spremiti cijele brojeve na sljedeći način:



U radnoj memoriji, varijable ***a*** i ***b*** "pokazuju" na dvije različite lokacije. Nakon što izvršimo cijeli program, ispisat će se različite vrijednosti:

```
// primjer:
let a = 5;
let b = a;

b = b + 2;

console.log(a); // 5
console.log(b); // 7
```

Dakle, ako se radi o vrijednosnim tipovima, kod pridruživanja ili slanja u funkcije/metode, šalje se vrijednost (ili kopija vrijednosti).

Ako uspoređujemo vrijednosne tipove podataka dobit ćemo sljedeće rezultate:

```
// uspoređivanje vrijednosnih:
let prvi = 1;
let prviKopija = 1;
console.log(prvi === prviKopija); // true
console.log(prvi === 1);       // true
console.log(prvi === prvi);     // true
```

Primjer slanja vrijednosnih tipova podataka u funkciju (***pass by value***):

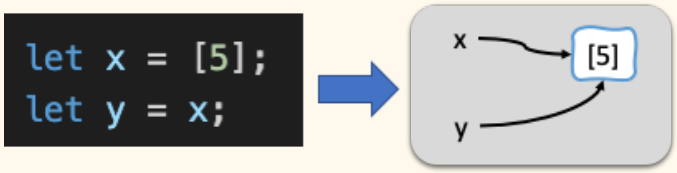
```
// slanje po vrijednosti u funkciju:
function kvadrat(a){
 a = a * a;
 return a;
}
```

```
let stranica = 10;
let rez = kvadrat(stranica);
console.log("stranica: " + stranica); // 10
console.log("rez: " + rez); // 100
```

Premda se vrijednost varijable ***a*** u funkciji *kvadrat* promijenila, ipak je originalna vrijednost varijable *stranica* ostala ista.

Referentni tipovi podataka

U sljedećem primjeru iskoristit ćemo jedan niz kao primjer referentnog tipa podatka. Definiramo ***x*** kao niz od jednog elementa:



Nakon izvršavanja sljedećeg programa u kojem mijenjamo samo **y** varijablu:

```
let x = [5];
let y = x;

y.push(2); // dodajemo jedan element u y
console.log(x); // [5, 2]
console.log(y); // [5, 2]
```

Ispisat će se iste vrijednosti za obje varijable ***x*** i ***y***!

Ako uspoređujemo referentne tipove, možemo primijetiti sljedeće:

```
// uspoređivanje referentnih:
let niz1 = [1];
let niz2 = [1];
console.log(niz1 === niz2); // false
console.log(niz1 === [1]);  // false

let niz3 = niz1;
console.log(niz1 === niz3); // true
console.log(niz1 === niz1);  // true
```

Varijable *niz1* i *niz2* imaju istru strukturu i sadržaj, ali su ipak različite jer su to dva niza na različitim lokacijama. Varijabla *niz3* "pokazuje" na istu lokaciju kao i *niz1* tako da je u tom slučaju rezultat uspoređivanja ***true*** jer se radi o istim nizovima.

Primjer slanja referetnih tipova u funkciju (***pass by reference***):

```
// slanje po referenci u funkciju:
function povecajBr(obj){
 obj.br++;
 console.log("Ispis u funkciji: " + obj.br); // 2
}

let a = {
 br: 1
}

console.log("Ispis prije funkcije: " + a.br); // 1
povecajBr(a);
console.log("Ispis nakon funkcije: " + a.br); // 2
```

Rezultat ispisa:

```
Ispis prije funkcije: 1
Ispis u funkciji: 2
Ispis nakon funkcije: 2
```

Obzirom da je varijabla ***a*** zapravo objekt, to znači da je **referentnog tipa**. Vrijednost varijable se promijenila unutar funkcije te je ostala promijenjena i nakon što je funkcija završila.

Vrijednosni i referentni tipovi podataka spremaju se u radnoj memoriji na različite načine.

Klase u JavaScript-u

U ovoj vježbi ćemo se podsjetiti što je to klasa i vidjeti kako se piše u JavaScript-u. Klasa je korisnički definiran tip, a definiramo ga tako da pišemo ključnu riječ **class** iza kojeg slijedi naziv klase. Na prvi pogled, piše se isto kao u C#-u.

```
class Pas {
  //tijelo klase
}
```

Međutim, u ostatku se počinju razlikovati. U C#-u postoji konstruktor klase koji mora imati isti naziv kao i sama klasa kojoj pripada. Međutim, konstruktor u JavaScript-u piše se pomoću

ključne riječi **constructor**. Jedinstvena ključna riječ nam također sugerira da na taj način možemo definirati samo jedan konstruktor.

C#

```
class Pas {
  public Pas() {
    //konstruktor klase
  }
}
```

JavaScript

```
class Pas {
  constructor() {
    //konstruktor klase
  }
}
```

Po dogovoru ćemo i ovdje koristiti veliko početno slovo za naziv klase, kako bi ih lakše razlikovali od ostalih stvari (npr. varijabli). Za sve ćemo nastojati zadržati stil kojeg vidimo u ostalim, ugrađenim elementima JavaScript-a.

U C#-u razlikujemo polja i svojstva. Polja su varijable definirane unutar klase, a svojstva smo koristili kao neku vrstu zaštite i ograničenja pristupa poljima. Također, pisali smo ih pomoću **get** i **set**. JavaScript podržava koncept polja i svojstava no sintaksa je nešto drugačija. Obzirom da JavaScript također omogućuje puno više različitih pristupa i načina pisanja za neke stvari, ovdje ćemo koristiti ono što je najsličnije C#-u kako bi olakšali prijelaz između ta dva jezika.

Uzmimo prethodni primjer osobe. Dok smo imali jednu osobu, onda nije bilo teško napraviti objekt koji se sastojao od svih potrebnih polja (ime, dob, adresa i sl.). Međutim, čim počnemo pisati više od jedne osobe, kôd se počne duplicirati te imamo potrebu definirati kôd osobe na jednom mjestu. Prema tome, definirat ćemo klasu *Osoba* s odgovarajućim poljima i konstruktorom.

Polja klase i konstruktor

Polja klase su varijable u kojima su spremljeni podaci. U JavaScript-u ćemo ih definirati u konstruktoru, iza ključne riječi **this** i operatora točke:

```
class Osoba{
  constructor(){
    this.ime = "";
    this.prezime = "";
    this.dob = 0;
  }
}
```

Nije potrebno definirati tipove i ne koristimo ključnu riječ **let** za deklaraciju. Konstruktor može primati parametre pa možemo odmah postaviti ime i prezime osobe:

```
class Osoba{
  constructor(i, p){
    this.ime = i;
    this.prezime = p;
    this.dob = 0;
  }
}
```

Parametri koje konstruktor prima, namjerno su u gornjem primjeru drugačijih naziva od polja *ime* i *prezime* kako bi bilo jasno da se radi o različitim varijablama, ali mogu se zvati jednako.

Metode u klasama

Ponovimo, metode pišemo tako da najprije navedemo:

- naziv metode (pisat ćemo ih malim slovima, jer se tako pišu ostale stvari u JS),
- popis parametara u okruglim zagradama (ako ih ima),
- tijelo metode u vitičastim zagradama.

Primjer:

```
class Osoba{
  constructor(i, p){
    this.ime = i;
    this.prezime = p;
    this.dob = 0;
  }

  imePrezime(){
    return this.ime + " " + this.prezime;
  }

}
```

Primjer stvaranja dva objekta tipa *Osoba*:

```
let prva = new Osoba("Pero", "Perić");
let druga = new Osoba("Ana", "Anić");
```

Možemo ih ispisati na konzolu:

```
console.log(prva);
console.log(druga);
```

Provjerite kako izgleda ispis u konzoli!

Također, možemo ispisati samo ime i prezime pomoću metode koju smo napisali:

```
console.log(prva.imePrezime());
```

Canvas

JavaScript se ne koristi samo za rad s tekstom i brojevima već pomoću njega možemo i crtati. Crtat ćemo na HTML elementu koji se zove **canvas**. Element *canvas* možemo zamisliti kao prazan list papira na kojem možemo crtati npr. crte, geometrijske likove, slike, tekst…

Canvas je također glavni HTML element na kojeg ćemo koristiti u projektu. Piše se s početnim i završnim oznakama:

```
<canvas></canvas>
```

Kako bi ga lakše mogli identificirati u projektu, možemo mu dati ***id*** te postaviti željene dimenzije.

```
<canvas width="640" height="480" id="game"></canvas>
```

Ubuduće to neće biti potrebno raditi jer ćemo imati gotov okvir u kojem se canvas prilagođava veličini stranice.

Obzirom da se na canvasu prikazuje cijeli projekt, pokušat ćemo ga upoznati na jednostavnom okviru kojeg možete preuzeti sa stranice kolegija. Primjer okvira povezat će:

- klase koje smo naučili pisati,
- klase koje ćemo koristiti u projektu,
- petlju igre.

Jednostavni okvir sadrži sljedeće:

- Klasu ***Slika*** koja će se brinuti o slikama.
- Klasu ***Display*** koja će se baviti crtanjem na *canvas*.
- Klasu ***Sprite*** koja predstavlja lika kojeg crtamo.

U JavaScript-u postoji klasa ***Image*** koja omogućuje stvaranje HTML elementa <img> koji se inače koristi za prikaz slika. Prema tome, ne možemo koristiti taj naziv. Također, u klasi *Slika* su definirane još neke dodatne stvari koje će nam olakšati prve korake.

> **Važno:** Slike se na HTML stranici učitavaju nakon ostalih dijelova stranice.

Klasa *Slika* sadržava jedno polje koje je tipa *Image*. Nakon što se objektu tipa *Image* pridruži putanja do slike, ona se počinje učitavati. Međutim, učitavanje slike je sporo tako da program zapravo nastavlja dalje iako slika nije učitana. Iz tog razloga klasa *Slika* sadrži provjeru je li slika učitana.

Za crtanje nam je dodatno potreban JavaScript objekt koji sadrži metode za crtanje: *drawing context* kojeg uzimamo od *canvas* elementa pomoću metode ***getContext()***. Dvije naredbe koje su nam potrebne za dohvaćanje potrebnih stvari za crtanje nalaze se u konstruktoru klase ***Display***. Klasa *Display* također ima i metodu ***draw()*** koja omogućuje crtanje lika.

Klasa ***Sprite*** sadrži minimum potrebnih podataka kako bi je mogli nacrtati:

- koordinate: ***x*** i ***y***,
- širinu i visinu: ***width*** i ***height*** (pišemo ih na engleskom jer se često spominju, a također kako bi izbjegli **š** u nazivu),
- sliku tipa *Slika*.

```
class Sprite {
  constructor(slika, x, y, w, h) {
    this.x = x;
    this.y = y;
    this.width = w;
    this.height = h;

    this.slika = slika;
  }
}
```

U projekt su dodane dvije slike: *dog.png* i *cat.png* koje se nalaze u mapi *slike*. Slike su za potrebe ovog primjera preuzete sa stranice s besplatnim slikama (https://pixabay.com/).

cat.png

dog.png

Radi jednostavnosti, napravljen je objekt u kojeg smo spremili sve što treba za te slike:

```
let slike = {
  cat: new Slika("slike/cat.png"),
  dog: new Slika("slike/dog.png")
}
```

Također, prilikom instanciranja slika, dogodit će se i učitavanje slika tako da će slike biti spremne za rad dok stignemo pokrenuti ostatak programa.

Bit će nam potrebna i ***petlja igre*** koju ste upoznali na predavanjima. Petlja igre će se stalno pozivati i osvježavati *canvas*:

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

  window.requestAnimationFrame(gameLoop);
}
```

Objekt **window** kojeg vidimo u petlji je globalni objekt koji predstavlja prozor web preglednika. Svi globalni objekti, funkcije, konstante i varijable automatski su sadržane u objektu **window**.

Iz objekta **window** možemo doznati npr. dimenzije trenutnog *prozora*, možemo otvoriti ili zatvoriti prozor web preglednika i sl.

Metoda ***window.requestAnimationFrame(****callback****)*** šalje informaciju web pregledniku da želimo obaviti animaciju. Metoda prima ***callback*** parametar. Taj parametar predstavlja funkciju koja će obaviti animaciju prije sljedećeg iscrtavanja stranice, a u našem slučaju to je ***gameLoop()***. Potrebno je pozvati ovu metodu svaki put kad želimo osvježiti (*update*) animaciju. Funkcija *gameLoop* ima jedan argument kojeg prosljeđuje. To je vrijeme (*time stamp*) koje označava trenutak u kojem *requestAnimationFrame()* počinje izvršavati funkcije.

Također, metoda *requestAnimationFrame()* vraća jedinstveni ***id*** svakog zahtjeva za animacijom kako bi mogli zaustaviti animaciju, ali to sad nećemo raditi već ćemo samo osvježiti stranicu kad bude potrebno zaustaviti animaciju.

Kako animacija ne bi započela prije nego što su spremne slike, sve skupa će započeti klikom na “botun” kojim se pokreće funkcija:

```
function start() {
  setup();
  gameLoop();
}
```

U funkciji ***setup()*** instancirat ćemo likove koje koristimo u projektu. Npr.:

```
// globalne varijable
let cat = null;
```

```
function setup() {

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

}
```

U funkciji ***gameLoop()*** pozivamo crtanje lika *cat*:

```
display.draw(cat);
```

Zadaci

- Zadatak 1.
  - U funkciji ***setup()*** stvoriti mačku na poziciji (0, 0) dimenzija 50 x 50 px.
  - U funkciji ***gameLoop()*** pozvati crtanje mačke.
- Zadatak 2.
  - Dodati lik psa na nekoj drugoj poziciji.
  - Odredite sami poziciju i veličinu psa.
- Zadatak 3.
  - Napraviti metode za kretanje mačke po *canvas-u* za sve smjerove po 10 px: *moveRight(), moveLeft(), moveUp(), moveDown()* .
  - Pokrenite mačku iz konzole! Probajte sve smjerove.
  - Promijenite prethodne metode tako da primaju broj koraka.
- Zadatak 4
  - Napravite jednu metodu za kretanje koja:
    - Prima parametre: *broj koraka* i *smjer*.
    - Broj koraka je cijeli broj, a smjer je tekst (lijevo, desno, gore, dolje).
    - Ako se ne upiše smjer, mačka ide desno.
  - Iskoristite metode za kretanje iz prethodnog zadatka.
  - Testirajte rad nove metode pozivom iz konzole za različite smjerove te bez navođenja smjera.

_

Literatura

[1]
„JavaScript Methods“. https://www.w3schools.com/js/js_object_methods.asp (ožujak 2022.).

[2]
„Functions - JavaScript | MDN“. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions (ožujak 2022.).

[3]
A. Chiarelli, *Mastering JavaScript Object-Oriented Programming*. Packt, 2016. [Na internetu]. Dostupno na: https://www.packtpub.com/product/mastering-javascript-object-oriented-programming/9781785889103

[4]
D. Stančer, *Osnove JavaScripta*, C501 izd. Sveučilište u Zagrebu, Sveučilišni računski centar, 2015. [Na internetu]. Dostupno na: https://www.srce.unizg.hr/files/srce/docs/edu/osnovni-tecajevi/c501_polaznik.pdf

[5]
N. Morgan, *JavaScript for kids: A playful introduction to programming*. No Starch Press, 2014.