Univerzitet u Sarajevu
Elektrotehnički fakultet Sarajevo
Računarstvo i informatika
Predmet: Web tehnologije

Laboratorijska vježba br. 5: JavaScript (closure)

1. Vidljivost varijabli i closure

JavaScript je veoma funkcionalno orijentisan programski jezik. Ovo daje puno slobode. Funkcija se može kreirati u svakom trenutku, proslijediti kao argument drugoj funkciji, a zatim kasnije pozvati sa potpuno drugog mjesta u kodu. Već znamo da funkcija može pristupiti varijablama izvan nje (tzv. "vanjske" varijable).

Ali šta se dešava ako se vanjske varijable promjene od trenutka kreiranja funkcije? Hoće li funkcija dobiti nove ili stare vrijednosti? A šta ako je funkcija proslijeđena kao argument i pozvana s drugog mjesta koda, hoće li dobiti pristup vanjskim varijablama na novom mjestu?

U nastavku će detaljnije biti opisana vidljivost varijabli u JavaScript-u kako bi se razumjeli ovi i složeniji scenariji.

1.1. Blokovi koda

Ako je varijabla deklarirana unutar bloka koda `{...}`, vidljiva je samo unutar tog bloka. Primjer:

```
{
  // rukovanje lokalnim varijablama
  // koje ne trebaju biti vidljive izvana

  let message = "Hello"; // varijabla je vidljiva samo u ovom bloku

  alert(message); // Hello
}

alert(message); // Error: message is not defined
```

Ovo se može koristiti za izoliranje dijela koda koji radi svoj vlastiti zadatak, sa varijablama koje samo njemu pripadaju:

```
{
  // ispis poruke
  let message = "Hello";
  alert(message);
}

{
  // ispis neke druge poruke
  let message = "Goodbye";
  alert(message);
}
```

Da se ne koriste blokovi, došlo bi do greške

Obratite pažnju da bi bez odvojenih blokova došlo do greške ako koristimo `let` sa postojećim imenom varijable:

```
// ispis poruke
let message = "Hello";
alert(message);

// ispis neke druge poruke
// Error: Identifier 'message' has already been declared
let message = "Goodbye";
alert(message);
```

Kada je riječ o `if`, `for`, `while` i slično, varijable deklarirane u sklopu `{...}` su vidljive samo unutar tog bloka:

```
if (true) {
  let phrase = "Hello!";

  alert(phrase); // Hello!
}

alert(phrase); // Error, phrase is not defined
```

U prethodnom isječku koda, nakon što se završi `if`, drugi `alert` neće vidjeti varijablu `phrase`, zbog čega se javlja greška.

Ovo je korisno, jer nam omogućava kreiranje blok-lokalnih varijabli.

Slična stvar vrijedi za `for` i `while` petlje:

```
for (let i = 0; i < 3; i++) {
  // varijabla i je jedino vidljiva unutar ove for petlje
  alert(i); // ispisuje se 0, pa 1, pa 2
}

alert(i); // Error, i is not defined
```

Vizualno, `let i` je izvan bloka `{...}`. Međutim, `for` konstrukcija je specifična: varijabla, deklarirana unutar ove konstrukcije, se smatra dijelom bloka.

1.2. Ugniježđene funkcije

Funkcija se naziva "ugniježđena" (eng. *nested*) kada je kreirana unutar druge funkcije. Ovo je u JavaScript-u moguće uraditi na jednostavan način. Može se koristiti za organiziranje koda, poput:

```
function sayHiBye(firstName, lastName) {

  // pomocna ugnijezdjena funkcija koja se koristi u nastavku
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );
}
```

Ovdje je ugniježđena funkcija `getFullName()` napravljena radi praktičnosti. Ona može pristupiti vanjskim varijablama zbog čega je u stanju vratiti puno ime (tj. ime nastalo spajanjem imena `firstName` i prezimena `lastName`). Ugniježđene funkcije su prilično česte u JavaScript-u.

Mnogo je interesantnija činjenica da ugniježđena funkcija može biti vraćena: bilo kao svojstvo novog objekta ili kao rezultat sama po sebi. Nakon toga se može koristiti negdje drugo. Bez obzira gdje se koristi, i dalje ima pristup istim vanjskim varijablama.

U nastavku je prikazana funkcija `makeCounter` koja kreira funkciju koja vraća sljedeći broj pri svakom pozivu:

```
function makeCounter() {
  let count = 0;
```

```
  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Uprkos tome što je prethodni isječak koda jednostavan, neznatne modifikacije ovog koda imaju praktičnu upotrebu. Na primjer, mogu se koristiti kao generator slučajnih brojeva za generisanje slučajnih vrijednosti za automatizovane testove. Kako ovo funkcionira? Ako kreiramo više brojača, hoće li oni biti nezavisni? Šta se ovdje dešava sa varijablama?

Razumijevanje ovih stvari je bitno za cjelokupno poznavanje JavaScript-a i korisno je za složenije scenarije. Zbog toga će detaljnije biti opisane pojedinosti.

1.3. Leksičko okruženje

Radi jasnoće, objašnjenje je podijeljeno u više koraka.

1.3.1. Korak 1: Varijable

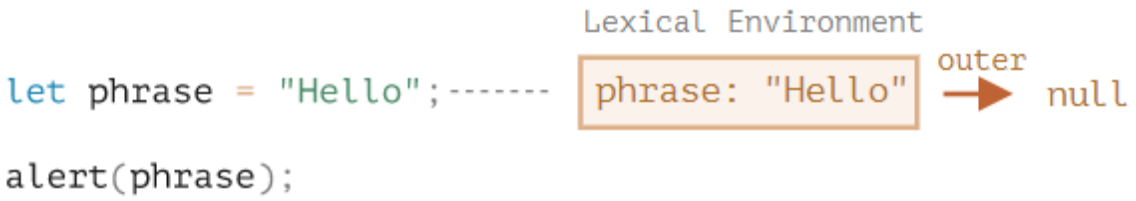
U JavaScript-u, svaka funkcija koja se izvršava, blok koda `{...}` i skripta u cjelini imaju interni (skriveni) pridruženi objekt poznat kao "Leksičko okruženje" (eng. *Lexical Environment*).

Objekt "Leksičko okruženje" se sastoji od dva dijela:

1. "Zapis okruženja" (eng. *Environment Record*) – objekt koji pohranjuje sve lokalne varijable kao svoja svojstva (i neke druge informacije kao što je vrijednost `this`).
2. Referenca na vanjsko leksičko okruženje, tj. ono povezano sa vanjskim kodom.

"Varijabla" je samo svojstvo posebnog internog objekta, "Zapisa okruženja". "Pristupiti vrijednosti varijable ili promijeniti varijablu" znači "pristupiti vrijednosti svojstva ili promijeniti svojstvo tog objekta".

Na slici 1.1. u jednostavnom primjeru koda bez funkcija, postoji samo jedno leksičko okruženje:

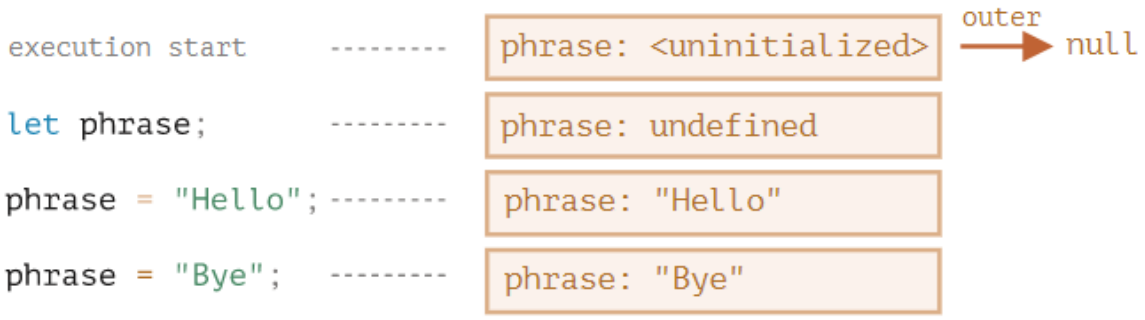


Slika 1.1. Primjer sa samo jednim leksičkim okruženjem

Ovo je takozvano globalno leksičko okruženje, povezano sa čitavom skriptom.

Na slici 1.1, pravougaonik označava *Environment Record* ("skladište" za varijable), a strelica označava vanjsku referencu. Globalno leksičko okruženje nema vanjsku referencu, zbog čega strelica pokazuje na `null`.

Kako se kod počinje izvršavati, mijenja se leksičko okruženje. U nastavku je prikazan nešto duži kod:



Slika 1.2. Leksičko okruženje za nešto složeniji kod

Na slici 1.2, pravougaonici na desnoj strani pokazuju kako se globalno leksičko okruženje mijenja tokom izvršenja:

- Kada se skripta pokrene, leksičko okruženje je unaprijed popunjeno svim deklariranim varijablama.
    - Na početku su ove varijable u "neinicijaliziranom" (eng. *uninitialized*) stanju. To je posebno interno stanje, koje označava da engine zna za varijablu, ali ne može biti referencirana dok se ne deklarira sa `let`. Gotovo je isto kao da varijabla ne postoji.
- Zatim se javlja definicija `let phrase`. Još uvijek nema dodjele, tako da je vrijednost ove varijable `undefined`. Ovu varijablu možemo koristiti od ovog trenutka nadalje.
- varijabli `phrase` je dodijeljena vrijednost.
- varijabla `phrase` mijenja vrijednost.

Za sada sve izgleda prilično jednostavno. Dakle:

- Varijabla je svojstvo posebnog internog objekta, povezanog sa blokom ili funkcijom ili skriptom koja se trenutno izvršava.

- Rad sa varijablama je zapravo rad sa svojstvima tog objekta.

Leksičko okruženje je specifikacijski objekt

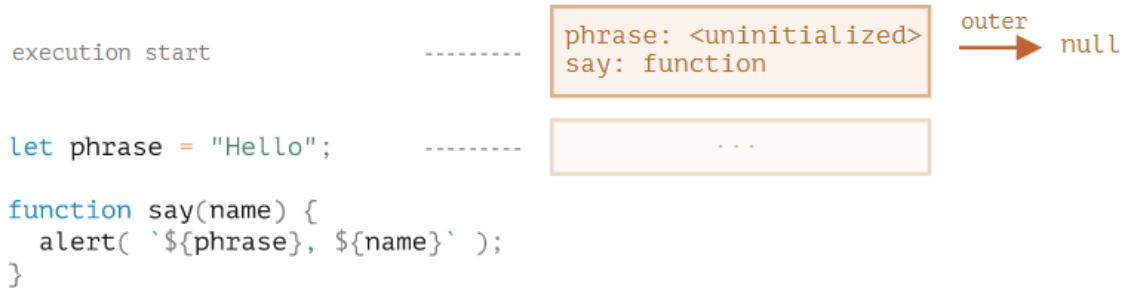
"Leksičko okruženje" je specifikacijski objekt, tj. postoji samo "teoretski" u specifikaciji jezika kako bi se opisalo kako stvari funkcioniraju. Ovaj objekt se ne može dobiti u našem kodu i ne može se njime direktno manipulirati.

JavaScript engine ga također mogu optimizirati, odbaciti varijable koje se ne koriste za uštedu memorije i izvesti druge interne trikove, sve dok vidljivo ponašanje ostaje onako kako je opisano.

1.3.2. Korak 2: Deklaracija funkcija

Funkcija je također vrijednost, poput varijable. Razlika je u tome što je deklaracija funkcije odmah potpuno inicijalizirana. Kada se kreira leksičko okruženje, deklaracija funkcije odmah postaje funkcija spremna za upotrebu.

Na primjer, u nastavku je prikazano početno stanje globalnog leksičkog okruženja kada dodamo funkciju:



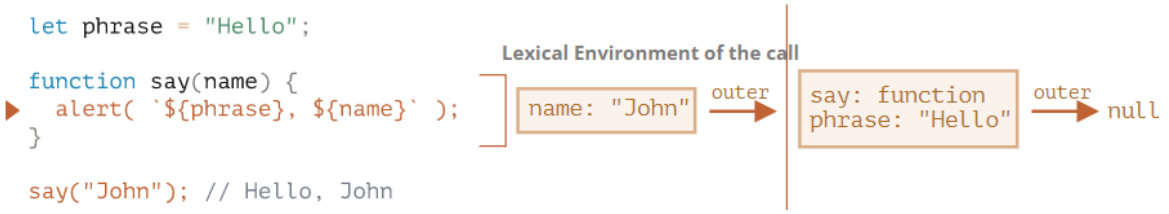
Slika 1.3. Početno stanje leksičkog okruženja kada se doda funkcija

Naravno, ovo ponašanje se odnosi samo na deklaracije funkcija, a ne na funkcijske izraze gdje dodjeljujemo funkciju varijabli, kao što je `let say = function(name)...`.

1.3.3. Unutrašnje i vanjsko leksičko okruženje

Kada se funkcija pokrene, na početku poziva, automatski se kreira novo leksičko okruženje za pohranjivanje lokalnih varijabli i parametara poziva.

Na primjer, za `say("John")`, to izgleda ovako (izvršenje je u liniji koda koja je označena strelicom):



Slika 1.4. Izgled leksičkog okruženja prilikom izvršenja funkcije

Tokom poziva funkcije postoje dva leksička okruženja: unutrašnje (za poziv funkcije) i vanjsko (globalno):

- Unutrašnje leksičko okruženje odgovara trenutnom izvršenju funkcije `say`. Ima jedno svojstvo: `name`, koje je argument funkcije. Pozvali smo funkciju sa `say("John")`, tako da varijabla `name` ima vrijednost `"John"`.
- Vanjsko leksičko okruženje je globalno leksičko okruženje. Ima varijablu `phrase` i samu funkciju.

Unutrašnje leksičko okruženje ima referencu na vanjsko, tj. na `outer`.

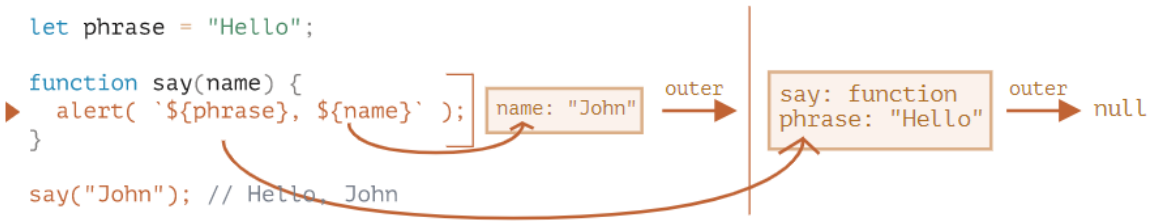
Kada kod želi pristupiti varijabli – prvo se pretražuje unutrašnje leksičko okruženje, zatim vanjsko, zatim ono izvan tog i tako sve do globalnog.

Ako se varijabla nigdje ne pronađe, javlja se greška u strogom (eng. *strict*) načinu rada (bez korištenja `use strict`, dodjela nepostojećoj varijabli stvara novu globalnu varijablu, radi kompatibilnosti sa starim kodom).

U primjeru sa slike 1.4., pretraga se odvija na sljedeći način:

- Za varijablu `name`, `alert` unutar funkcije `say` odmah pronalazi ovu varijablu u unutrašnjem leksičkom okruženju.
- Kada želi pristupiti varijabli `phrase`, pošto `phrase` ne postoji lokalno, prati se referenca na vanjsko leksičko okruženje i ova varijabla se tamo pronalazi.

Ovi koraci su ilustrirani na slici 1.5.



Slika 1.5. Postupak pretrage unutrašnjeg i vanjskog leksičkog okruženja

1.3.4. Funkcija kao povratna vrijednost

Vratimo se na primjer funkcije `makeCounter`.

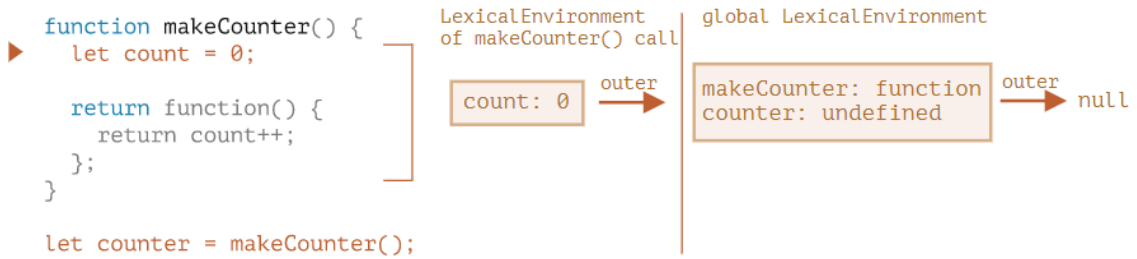
```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
```

Na početku svakog poziva funkcije `makeCounter()` kreira se novi objekt leksičkog okruženja za pohranjivanje varijabli za ovaj poziv funkcije `makeCounter`.

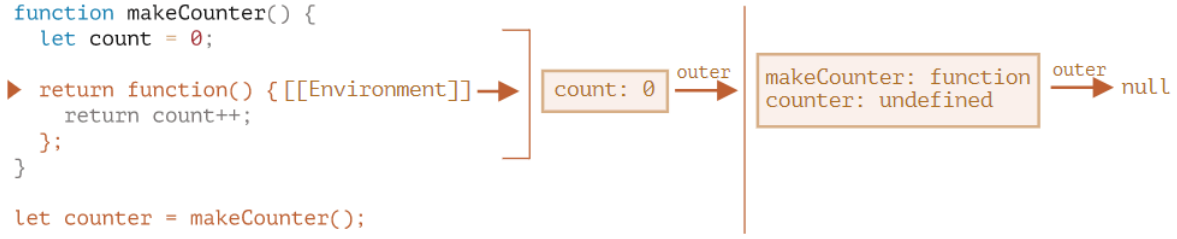
Dakle, postoje dva ugniježđena leksička okruženja, baš kao u prethodnom primjeru:



Slika 1.6. Leksičko okruženje funkcije makeCounter

Drugačije je to što se tokom izvršavanja `makeCounter()` kreira kratka ugniježđena funkcija od samo jedne linije koda: `return count++`. Još uvijek ne pokrećemo ovu funkciju, već je samo kreiramo.

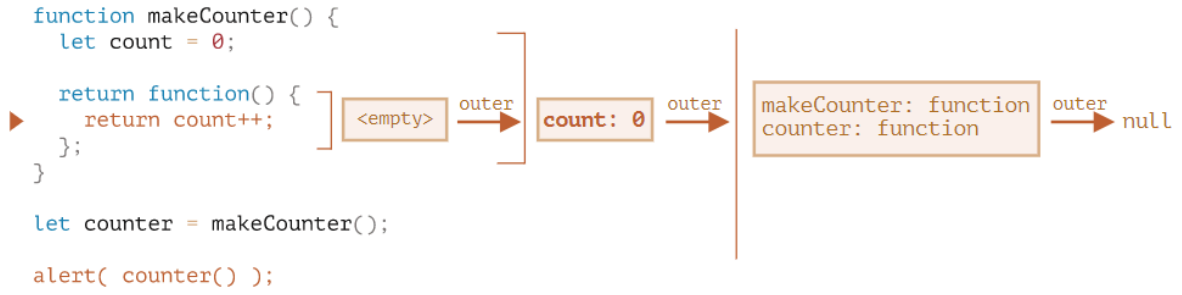
Sve funkcije pamte leksičko okruženje u kojem su napravljene. Dakle, sve funkcije imaju skriveno svojstvo pod nazivom `[[Environment]]`, koje zadržava referencu na leksičko okruženje u kojem je funkcija kreirana (slika 1.7.).



Slika 1.7. Skriveno svojstvo [[Environment]]

Dakle, `counter.[[Environment]]` ima referencu na `{count: 0}` leksičko okruženje. Na ovaj način funkcija pamti gdje je kreirana, bez obzira na to gdje je pozvana. Referenca `[[Environment]]` se postavlja jednom i zauvijek u trenutku kreiranja funkcije.

Kasnije, kada se pozove `counter()`, kreira se novo leksičko okruženje za poziv funkcije, a njegova referenca na vanjsko leksičko okruženje se uzima iz `counter.[[Environment]]` (slika 1.8.).

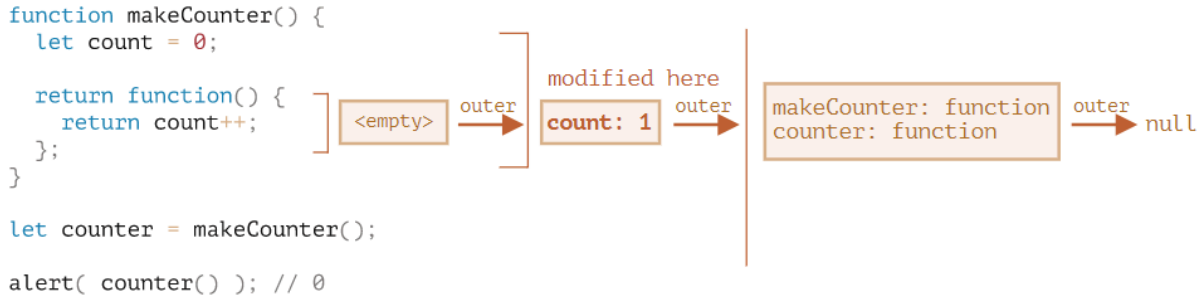


Slika 1.8. Funkcija pamti leksičko okruženje gdje je kreirana, bez obzira na to gdje je pozvana

Sada kada kod unutar `counter()` traži varijablu `count`, prvo pretražuje svoje vlastito leksičko okruženje (ono je prazno, jer tamo nema lokalnih varijabli), zatim leksičko okruženje poziva vanjske `makeCounter()` funkcije, gdje se pronalazi varijabla `count` i vrši se njena izmjena.

Varijabla se ažurira u leksičkom okruženju u kojem živi.

Na slici 1.9. je ilustrirano stanje nakon izvršenja funkcije:



Slika 1.9. Stanje nakon izvršenja makeCounter() funkcije

Ako više puta pozovemo funkciju `counter()`, varijabla `count` će se povećati na vrijednost 2, pa na vrijednost 3 i tako dalje, na istom mjestu.

1.4. *Closure*

U programiranju postoji opći termin "closure", koji bi programeri općenito trebali znati.

Closure je funkcija koja pamti svoje vanjske varijable i može im pristupiti. U nekim jezicima to nije moguće, ili bi funkcija trebala biti napisana na poseban način da se to ostvari. Ali kao

što je prethodno objašnjeno, u JavaScript-u su sve funkcije po svojoj prirodi "closures" (postoji samo jedan izuzetak, koji se javlja kada se koristi sintaksa `new Function` koju nećemo detaljnije analizirati). Dakle, *closure* se dešava kada funkcija zapamti svoje leksičko okruženje čak i kada se izvršava u drugom leksičkom okruženju.

Dakle, funkcije automatski pamte gdje su kreirane koristeći skriveno `[[Environment]]` svojstvo, a zatim njihov kod može pristupiti vanjskim varijablama.

1.4.1. Upotreba *closure*-a

Ukoliko pravite neki modul koji dijeli nekoliko varijabli i koji posjeduje nekoliko metoda, tada je korisno definirati glavnu funkciju modul koja će predstavljati zaštićeni *scope* i koja će kao rezultat vratiti objekat koji sadrži sve funkcije datog modula. Na ovaj način, sakrivene varijable modula su zaštićene od izmjena izvan modula, te se možete pouzdati da ćete uvijek čitati variable koje je modul postavio, a ne neki drugi kod koji koristi modul.

U nastavku je prikazan primjer koji koristi *module pattern*. Napišimo modul koji sadrži 5 metoda: *saberi*, *oduzmi*, *sx* (skalarni proizvod), *postaviX1* (postavlja prvi operand operacije) i *ispisi*.

```
let Operacije = function(){
    let operand1 = {x:0,y:0};
    const saberi = function(operand2){
      operand1 = {x:operand1.x+operand2.x,y:operand1.y+operand2.y};
    }
    const oduzmi = function(operand2){
      operand1 = {x:operand1.x-operand2.x,y:operand1.y-operand2.y};
    }
    const sx = function(operand2){
      return operand1.x*operand2.x + operand1.y*operand2.y;
    }
    const postaviX1 = function(X1){
      operand1 = X1;
    }
    const ispisi = function(){
      console.log("x: "+operand1.x+" , y: "+operand1.y);
    }
    return {
      saberi: saberi,
      oduzmi: oduzmi,
      sx: sx,
      postaviX1: postaviX1,
```

```
      ispisi: ispisi
  }
};

//primjer koristenja modula
let op=Operacije();
op.postaviX1({x:1,y:0});
op.ispisi();
let operand1={x:1,y:1};
op.saberi({x:2,y:2});
op.ispisi();
```

U primjeru je definiran modul *Operacije*. Za ovaj modul vrijedi da su sve njegove varijable i funkcije definirane unutar ograničenog *scope*-a, tj. postoji jedna funkcija koja okružuje kod modula. Ova funkcija se odmah poziva nakon definiranja i anonimna je (bez imena). Na ovaj način modul *Operacije* ima svoje vlastito leksičko okruženje u kojem su varijable zaštićene od vanjskih izmjena. Povratni tip ove funkcije je objekat koji sadrži kao atribute sve funkcije modula koje trebaju biti javne. Sve funkcije koje su definirane unutar modula, a nisu vraćene kroz objekat su privatne funkcije tog modula. Ovaj *pattern* nam omogućava da izbjegnemo konflikte prilikom imenovanja varijabli.

2. Zadaci

Zadatke označene **zelenom bojom** je potrebno uraditi u toku laboratorijske vježbe. Ukoliko studenti tokom laboratorijske vježbe ne urade te zadatke, onda kući (**najkasnije 8 sati** prije početka naredne laboratorijske vježbe) moraju uraditi i zadatke označene **zelenom bojom** i zadatke označene **narandžastom bojom**. Bez obzira da li se zadaci rade na vježbi ili kod kuće, **obavezno** ih je postaviti na odgovarajući *Bitbucket* repozitorij za vježbe.

Zadatak 1. Napišite funkciju *dodajDva* koja prima jedan parametar (pretpostavite da je parametar numeričkog tipa) i vraća vrijednost proslijeđenog parametra uvećanu za 2. Primjer poziva funkcije je prikazan u nastavku:

```
console.log(dodajDva(1)); // treba vratiti 3
console.log(dodajDva(2)); // treba vratiti 4
console.log(dodajDva(10)); // treba vratiti 12
```

Nakon toga, napišite funkciju *jednom* koja kao parametar prima tzv. *callback funkciju* (inače, *callback funkcija* je funkcija koja se prosljeđuje drugoj funkciji kao parametar i koja se onda poziva unutar nje kako bi obavila određenu radnju) i vraća funkciju. Kada se vraćena funkcija prvi put pozove, onda treba pozvati *callback funkciju* i vratiti odgovarajući izlaz. Ako se funkcija ponovo poziva više puta, onda umjesto ponovnog poziva *callback*-a, treba se vratiti samo vrijednost prvog poziva. Primjer poziva ove funkcije, gdje se kao *callback* koristi funkcija *dodajDva*, je prikazan u nastavku:

```
function jednom() {
// napisati tijelo funkcije
}

const jednomFunkcija = jednom(dodajDva);
console.log(jednomFunkcija (4)); // treba ispisati 6
console.log(jednomFunkcija (10)); // treba ispisati 6
console.log(jednomFunkcija (9001)); // treba ispisati 6
```

Zadatak 2. Neka je funkcija *dodaj* definirana na sljedeći način:

```
let dodaj = (function () {
    let brojac = 0;
    return function () {
       return brojac += 1;
    }
})();
```

Modificirajte prethodni kod tako da napišete modul koji definira brojački objekat, koji ima dvije metode: *dodaj()* i *resetuj()*. Pozivom metode *dodaj()*, vrijednost brojača se uveća za jedan. dok metoda *reset()* postavlja brojač na vrijednost 0.