Funkcije unutar funkcija i Closure

Još jedna od specijalnih osobina JavaScript jezika odnosi se na mogućnost kreiranja funkcija unutar funkcija. Takvu osobinu već ste mogli da vidite u lekcijama za nama – pogotovu prilikom korišćenja konstruktorskih funkcija. Ovakva mogućnost JavaScript jezika obezbeđuje realizaciju veoma naprednih ponašanja, o kojima će biti reči u lekciji koja je pred vama.

Na početku lekcije prvo će biti dat kratak pregled oblasti važenja JavaScript promenljivih iz perspektive funkcija. Iako je o tome bilo reči već nekoliko puta, u pitanju su veoma važne činjenice, čije razumevanje je neophodno za preostale, nove teme koje će biti obrađene u ovoj lekciji – pre svega za pojam Closure.

Function scope

Prilikom kreiranja JavaScript funkcije, ona postaje jedna oblast važenja, s obzirom na to da je reč o bloku koda. Takva oblast se drugačija naziva *Function scope* ili oblast važenja jedne funkcije. Oblast važenja jedne funkcije poseban značaj ima za promenljive koje se deklarišu unutar, ali i izvan takvih funkcija. Oblast važenja `var` promenljivih upravo je jedna funkcija, dok je kod `let` promenljivih to bilo koji blok koda.

Kada se promenljiva deklariše izvan funkcije, takva promenljiva postaje globalna promenljiva. Sa druge strane, kada se promenljiva deklariše unutar jedne funkcije, ona je vidljiva samo unutar takve funkcije, a drugačije se naziva lokalna promenljiva.

Sledeći primer bavi se ovom problematikom:

```
function myFunction() {
      var x = 4;
}
console.log(x);
```

Unutar funkcije `myFunction()`, deklariše se i inicijalizuje promenljiva sa nazivom `x` i vrednošću `4`. Izvan funkcije pokušava se ispisivanje vrednosti promenljive `x`. Primer proizvodi sledeći rezultat:

```
ReferenceError: x is not defined
```

Naravno, kod proizvodi grešku, zato što promenljiva `x` nije vidljiva na globalnom nivou, već samo unutar funkcije `myFunction()`.

Sa druge strane, globalna promenljiva je vidljiva unutar bilo koje funkcije ili bloka:

```
var x = 4;

function myFunction() {
      console.log(x)
}

myFunction();
```

Promenljiva x je globalna promenljiva i kao takva dostupna je i unutar funkcije myFunction(). Zato primer bez problema ispisuje vrednost promenljive x:

```
4
```

Svi dosad prikazani primeri stvorili bi identičan efekat i kada bi promenljive bile deklarisane korišćenjem ključne reči let.

Implicitno globalne promenljive

Veoma zanimljiva osobina JavaScript jezika jeste mogućnost kreiranja globalnih promenljivih, i to iz funkcija. Naime, dovoljno je promenljivu kreirati bez ključne reči var ili let i ona će postati globalna. Drugim rečima, inicijalizovanje promenljive koja prethodno nije deklarisana automatski takvu promenljivu čini globalnom. To ilustruje sledeći primer:

```
function myFunction() {
      x = 4;
}

myFunction();

console.log(x);
```

Unutar funkcije myFunction() inicijalizovana je promenljiva sa nazivom x, koja prethodno nije deklarisana. Ukoliko iskoristimo ono što smo dosad naučili, logično je zaključiti da će ovakav kod proizvesti grešku, zato što se pokušava pristupiti promenljivoj koja se nalazi unutar funkcije, a da pritom nije ni deklarisana. Ipak, efekat je potpuno drugačiji, pa na ovaj način promenljiva x postaje globalna. Naravno, to se događa u onom trenutku kada se takva linija izvrši, odnosno kada se pozove funkcija myFunction(). U primeru se ova funkcija poziva, a zatim se ispisuje vrednost promenljive x i na izlazu se dobija:

```
4
```

Ovako kreirana globalna promenljiva naziva se implicitna globalna promenljiva.

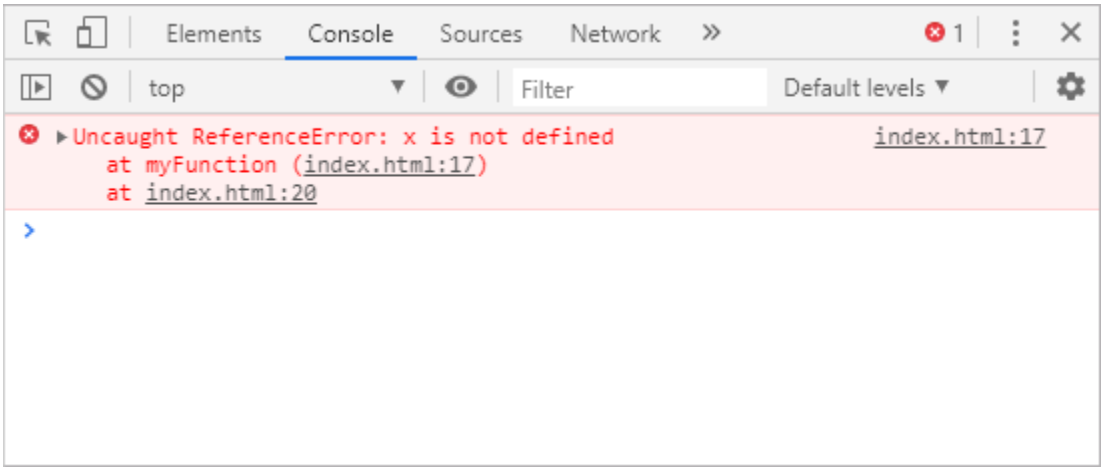
Napomena

Implicitne globalne promenljive su još jedna od osobina JavaScripta koje su dosta kritikovane. Gotovo da ne postoji razlog za korišćenje ovakve osobine jezika. Naime, ukoliko je potrebno deklarisati globalnu promenljivu, to je najbolje uraditi izvan svih blokova i funkcija, korišćenjem ključnih reči let ili var. Veoma često se upravo ilustrovana situacija u kodu pojavi potpuno nesvesno, odnosno programer greškom obavi inicijalizaciju promenljive koja ne postoji na globalnom nivou. Umesto da dođe do greške, izvršavanje se obavlja bez ikakvih problema, zato što greškom inicijalizovana promenljiva postaje globalna.

Sve to može stvoriti brojne logičke probleme prilikom izvršavanja koda, koje je veoma teško detektovati. Upravo zbog toga, novije verzije ECMAScript specifikacije poznaju takozvani **striktni mod**, koji neke od ovakvih nelogičnosti ispravlja. Na početak skripte dovoljno je postaviti 'use strict':

```
'use strict';
function myFunction() {
    x = 4;
}
myFunction();
```

Sada će ovakav kod proizvesti grešku (slika 6.1).



Slika 6.1. Greška prilikom izvršavanja koda u striktnom modu

Samopozivajuće funkcije

Samopozivajuće funkcije aktiviraju se automatski onoga trenutka kada izvršavanje koda dođe do njih, bez potrebe da budu eksplicitno pozvane. Primer jedne takve funkcije je sledeći:

```
(function () {
    console.log("Hello from self-invoking function");
})();
```

Kada izvršavanje koda dođe do ovako definisane funkcije, ona će odmah biti izvršena.

Kao logično, sada može postaviti pitanje: *koja je svrha samopozivajućih funkcija?*

Samopozivajuće funkcije se veoma često koriste kako bi se kreirao jedan zatvoreni opseg važenja promenljivih. Nešto ranije je rečeno da sve promenljive deklarisane unutar jedne funkcije postoje samo unutar takve funkcije. Takva osobina, u kombinaciji sa mogućnošću samopozivanja funkcija, može se iskoristiti za postizanje vrlo moćnih ponašanja.

Prilikom kreiranja ozbiljnijih sajtova i aplikacija, veoma lako može doći do konflikata u imenovanju promenljivih. Na primer, kreiranje sajtova i aplikacija veoma često podrazumeva korišćenje nekoliko različitih nezavisnih delova JavaScript koda, odnosno JavaScript biblioteka. Veoma lako se može desiti da se unutar različitih delova JavaScript koda upotrebi identičan identifikator za imenovanje promenljivih, funkcija ili objekata. Takva situacija može izazvati pojavu logičkih grešaka prilikom izvršavanja koda zbog konflikta imena. Kako bi se prevazišli ovakvi problemi, korišćenjem samopozivajućih funkcija moguće je kompletan kod jedne funkcionalnosti smestiti unutar jedne oblasti važenja:

```
(function () {
    var number = 5;
    console.log(number);
})();

console.log(number);
```

Ovakav kod proizvešće sledeći efekat:

```
5
Uncaught ReferenceError: number is not defined
```

Unutar samopozivajuće funkcije, promenljiva `number` poseduje vrednost 5. Ipak, izvan funkcije, promenljiva sa nazivom `number` uopšte ne postoji, što se jasno može zaključiti na osnovu dobijene greške. To znači da je izvan samopozivajuće funkcije moguće deklarisati promenljivu sa nazivom `number`, bez ikakve bojazni da će doći do konflikta imena.

Module dizajn šablon

Samopozivajuće funkcije su jedan od osnovnih elemenata pomoću kojih se u JavaScriptu realizuje Module dizajn šablon. S obzirom na to da Module dizajn šablon definiše postojanje nezavisnih programskih celina, sa privatnim prostorom imena, samopozivajuće funkcije idealan su kandidat za postizanje takvog ponašanja. Ovde se priča o samopozivajućim funkcijama ne završava, već će one u nastavku lekcije biti iskorišćene za praktičnu realizaciju još nekih naprednih JavaScript primera.

Funkcije unutar funkcija

Potpuno je legitimno pisati jednu funkciju unutar druge. Na taj način, unutrašnja funkcija je privatni član spoljne funkcije i ne može joj se pristupiti na globalnom nivou. Sledeći primer ilustruje funkciju unutar funkcije:

```
function subtractSquares(a,b) {
      function square(x) {
            return x * x;
      }
      return square(a) - square(b);
}

console.log(subtractSquares(6,3));
```

Primer ilustruje funkcionalnost za oduzimanje kvadrata dva broja. Naziv funkcije kojom se tako nešto obavlja je `subtractSquares` i ona prihvata dva parametra. Unutar ove funkcije definisana je još jedna funkcija, `square()`, čija je uloga podizanje broja na kvadrat. Ova funkcija se koristi interno, unutar funkcije `subtractSquares()`. Na kraju, funkcija `subtractSquares()`upošljava se vrednostima 6 i 3 i dobija se sledeći ispis:

```
27
```

Već je rečeno da se funkcija unutar funkcije ne može pozvati na globalnom nivou. Zato sledeći kod proizvodi grešku:

```
function subtractSquares(a,b) {
      function square(x) {
            return x * x;
      }
      return square(a) - square(b);
}

console.log(square(5));
```

Rezultat:

```
ReferenceError: square is not defined
```

Closure

Većina programskih jezika koji kao i JavaScript poseduju funkcije prvog reda poznaje i pojam Closure. Reč je zapravo o još jednom nazivu za osobinu JavaScript funkcija koju smo mi već videli u dosadašnjem toku ove lekcije. Pojam Closure odnosi se na spoj funkcije i pripadajuće leksičke oblasti unutar koje je takva funkcija definisana. Drugim rečima, Closure je funkcija zajedno sa referencama na sve elemente koji su definisani unutar sklopa u kome je definisana i ona sama.

Kreiranjem JavaScript funkcije ujedno se kreira i Closure. Kako biste na pravi način mogli da razumete šta nama ovakva osobina JavaScript jezika omogućava, u nastavku će biti prikazano nekoliko primera upotrebe Closurea.

Closure – primer 1

```
function generateCoupon(firstName, lastName){
    var code = (Math.random() * 10e16).toString();
    //send code, firstName and lastName over the network

    return "Hello " + firstName + " " + lastName + ". Your coupon
has been generated.";
}
```

Prikazanu funkciju možete doživeti kao neku uprošćenu verziju funkcionalnosti za generisanje nagradnih kupona. Logika je u potpunosti uprošćena tako da sadrži samo elemente koji su u ovom trenutku nama interesantni. Unutar funkcije, prvo se obavlja generisanje koda kupona. Zatim je postavljen komentar koji simulira programsku logiku, koja generisani kod, ime i prezime prosleđuje nekom udaljenom serveru na obradu. Tim se logika funkcije završava, a korisniku se vraća potvrdna poruka da je kupon uspešno kreiran:

```
generateCoupon("John", "Lord"); //Hello John Lord. Your coupon has been
generated.
```

Naredba ilustruje poziv funkcije `generateCoupon()` i povratnu vrednost koja se tom prilikom dobija. Ipak, šta ukoliko želimo da izvan funkcije `generateCoupon()` dođemo do vrednosti lokalne promenljive `code` koja se nalazi unutar funkcije:

```
function generateCoupon(firstName, lastName){
    var code = (Math.random() * 10e16).toString();
    //send code, firstName and lastName over the network

    return "Hello " + firstName + " " + lastName + ". Your coupon
has been generated.";
}

console.log(code);
```

Naravno, ovakav pokušaj pristupa promenljivoj `code` rezultuje generisanjem izuzetka:

```
Uncaught ReferenceError: code is not defined
```

Pojava izuzetka je potpuno logična. Promenljiva `code` je lokalna promenljiva funkcije `generateCoupon()`, pa samim tim nije vidljiva izvan funkcije. Ukoliko je izvan funkcije potrebno dobiti vrednost neke promenljive koja je za takvu funkciju lokalna, kao ispomoć se može iskoristiti Closure, odnosno osobina JavaScript funkcija da imaju pristup elementima sklopa u kome su definisane.

Za demonstraciju Closurea, prikazani primer za generisanje kupona biće redefinisan, ali ovoga puta na nešto ozbiljniji način, korišćenjem principa objektno orijentisanog programiranja. Na taj način, kuponi će biti modelovani kao JavaScript objekti koji se kreiraju korišćenjem sledeće konstruktorske funkcije:

```
function Coupon(firstName, lastName) {
    var code =  (Math.random() * 10e16).toString();
    var date = new Date();

    this.firstName = firstName;
    this.lastName = lastName;

}
```

Konstruktorska funkcija `Coupon()` koristiće se za kreiranje objekata kupona. Njoj se za kreiranje objekata prosleđuju dva parametra – `firstName` i `lastName`. Prosleđene vrednosti se postavljaju za vrednosti objektnih svojstava `firstName` i `lastName`. Ipak, pored ovih svojstava, konstruktorska funkcija poseduje i dva svojstva (`code` i `date`) čije se vrednosti automatski generišu unutar konstruktorske funkcije. Ipak, najznačajnija razlika između svojstava `code` i `date` i `firstName` i `lastName` ogleda se u upotrebi ključne reči `this`. Naime, svojstva `code` i `date` nisu definisana nad konkretnim instancama, zato što se ispred njih ne koristi ključna reč `this`. Upravo zbog toga se ovim svojstvima neće moći pristupiti nad kreiranim objektima:

```
let coupon1 = new Coupon("John", "Dean");
console.log(coupon1.code);
console.log(coupon1.date);
```

Sada je kreiran objekat koji predstavlja jedan kupon. Zatim se pokušava pristupiti svojstvima `code` i `date`. Unutar konzole se dobija:

```
undefined
undefined
```

Jasno je da na ovaj način nije moguće pristupiti svojstvima `code` i `date`. Ipak, za dobijanje vrednosti ovakvih svojstava može se iskoristiti Closure:

```
function Coupon(firstName, lastName) {
    var code =  (Math.random() * 10e16).toString();
    var date = new Date();

    this.firstName = firstName;
    this.lastName = lastName;

    this.getCode = function () { return code }
}
```

Anonimna funkcija koje je dodeljena svojstvu `getCode` jeste primer jednog Closurea. Reč je o funkciji koja se nalazi unutar druge funkcije – anonimna funkcija koja predstavlja Closure nalazi se unutar funkcije `Coupon()`. Kako bi ovakav Closure zapravo bio od neke koristi, on je dodeljen promenljivoj `getCode` koju će imati svi objekti koji se kreiraju korišćenjem `Coupon()` konstruktorske funkcije. Na taj način je jedna inače lokalna funkcija postala dostupna spoljašnjem svetu, odnosno, da budemo precizniji, sklopu izvan ovakve funkcije. Izvan funkcije `Coupon()` ne može se pristupiti lokalnoj promenljivoj `code`, ali njoj može pristupiti upravo kreirana funkcija (Closure), koja je izložena spoljnom svetu. Stoga je moguće napisati nešto ovako:

```
let coupon1 = new Coupon("John", "Dean");
console.log(coupon1.getCode());
```

Sada se unutar konzole dobija vrednost generisanog koda:

```
63379387351092830
```

Closure nam je u prikazanom primeru omogućio da uradimo nešto što bez njegove upotrebe nije bilo moguće uraditi – pristup lokalnom svojstvu jedne funkcije iz spoljašnjeg sklopa. Na ovaj način Closurei omogućavaju postizanje još bolje enkapsulacije. Naime, mi smo uspeli da unutar konstruktorske funkcije `Coupon()` kreiramo jedno privatno svojstvo. Pored toga, vrednost takvog privatnog svojstva korišćenjem Closurea moguće je samo čitati. Ovo je i potpuno logično, zato što jednom generisani kod kupona ne bi bilo dobro naknadno menjati.

Closure – primer 2

Naredni primer će možda na još bolji način dočarati kolika je moć Closurea:

```
let sayHello = (function () {

    const greeting = 'Hello ';

    return {
        to: function (name) {
            return greeting + name;
        }
    }

})();


let greeting = sayHello.to("John");

console.log(greeting);
```

Često se kaže da Closurei omogućavaju elementima da nadžive funkcije, odnosno objekte unutar kojih su definisani. Upravo takvu situaciju mi sada imamo u prikazanom primeru.

Za realizaciju primera iskorišćeno je nekoliko različitih mogućnosti JavaScript jezika o kojima je bilo reči u dosadašnjem toku kursa. U primeru je prvo definisana jedna anonimna, samopozivajuća funkcija, čiji se rezultat dodeljuje promenljivoj `sayHello`. Stoga, odmah nakon dolaska izvršnog okruženja do ovakvog koda, izvršavanje anonimne funkcije započinje. Unutar anonimne funkcije definisana je jedna konstanta sa nazivom `greeting` i vrednošću *Hello*. Pored ove konstante, unutar anonimne funkcije definisana je i povratna vrednost. Povratna vrednost je zapravo jedan objekat, kreiran korišćenjem objektnog literala. Takav objekat poseduje jednu funkciju. Reč je opet o anonimnoj funkciji koja je dodeljena svojstvu `to`. Funkcija prihvata jedan parametar (`name`), a zatim unutar svog tela formira poruku, za šta koristi vrednost konstante `greeting` definisane nešto ranije. Nakon ove anonimne funkcije, čiji se rezultat smešta unutar promenljive `sayHello`, obavlja se pozivanje metode `to()` nad takvom promenljivom. Metodi `to()` se prosleđuje ime (*John*), a zatim se dobijena povratna vrednost štampa unutar konzole:

```
Hello John
```

Kako izgleda tok izvršavanja upravo prikazanog primera?

Izvršavanje anonimne funkcije završiće se emitovanjem povratne vrednosti, koja će biti smeštena unutar promenljive `sayHello`. Povratna vrednost anonimne funkcije jeste objekat sa `to()` metodom. Takva metoda je klasičan primer Closurea, koji omogućava konstanti `greeting` da nadživi funkciju unutar koje je definisana.

U liniji u kojoj se poziva metoda `to()`, anonimna funkcija sa `greeting` konstantom je već završila svoju logiku. Ipak, kao što je rečeno, zbog postojanja pojma Closure, konstanta `greeting` ostaje efikasno zarobljena u zapamćenom okruženju funkcije `to()`. Imajte na umu da ovako nešto ne bi bilo moguće da funkcija `to()` na neki način nije uspela da izađe iz okvira funkcije u kojoj je definisana. To je u primeru postignuto tako što je objekat sa funkcijom `to()` emitovan kao povratna vrednost anonimne, samopozivajuće funkcije.

Module dizajn šablon

Upravo prikazani Closure oslikava još jedan klasičan primer korišćenja Module dizajn šablona. Korišćenjem samopozivajuće funkcije, efikasno je napravljen jedan zapečaćeni prostor imena. To sa najbolje može videti iz činjenice da spoljašnje okruženje ni na koji način ne može da direktno pristupi konstanti `greeting`. Zbog toga se kaže da je promenljiva `sayHello` zapravo referenca na jedan modul.

Emitovanje funkcije kao povratne vrednosti

Značajna osobina JavaScript funkcija odnosi se na mogućnost emitovanja jedne funkcije, kao povratne vrednosti neke druge funkcije. Emitovanjem funkcija kao povratnih vrednosti mogu se dobiti vrlo zanimljivi efekti. Na primer, evo kako se može obaviti sabiranje dva broja korišćenjem dve funkcije, pri čemu jedna drugu koristi kao svoju povratnu vrednost:

```
function sum(a) {

    return function (b) {
        return a + b;
    };

}
```

Kod ilustruje funkciju koja nosi naziv `sum` i prihvata jedan parametar(`a`). Ipak, ona kao svoju povratnu vrednost ima jednu drugu, anonimnu funkciju. Takva anonimna funkcija takođe prihvata jedan parametar (`b`), a unutar svoga tela obavlja sabiranje vrednosti `a` i `b` i emitovanje povratne vrednosti.

Prikazani primer pred nas postavlja puno novih tema za razgovor. Za početak, kako pozvati funkciju `sum()`, odnosno kako obaviti sabiranje dva broja korišćenjem ovakvog koda? S obzirom na to da prva funkcija vraća drugu i da obe prihvataju po jedan parametar, može se napisati:

```
let result = sum(5)(3);
```

Povratna vrednost ovakvog poziva će biti 8, odnosno parametri koji su prosleđeni pojedinačnim funkcijama biće sabrani, a vrednost dodeljena promenljivoj `result`. Ukoliko vam je i dalje teško da razumete na koji način ovaj primer funkcioniše, on će sada biti raščlanjen:

```
let sum2 = sum(5);
let result = sum2(3);
```

Umesto direktnog pozivanja povratne vrednosti prve funkcije, sada je primer razdvojen u dve naredbe, kako biste mogli lakše da vizuelizujete tok izvršavanja koda.

Kada je prikazani primer u pitanju, neophodno je da primetite još nešto. Svakoj od funkcija prosleđuje se po jedan parametar, a zatim se oba koriste unutar druge, ugnežđene, anonimne funkcije. Drugim rečima, funkcija koja se u primeru emituje kao povratna vrednost ima pristup parametru koji se prosleđuje nadfunkciji – `sum()`. Stoga, i ovaj primer ilustruje Closure osobinu JavaScript funkcija.

Funkcije višeg reda – Higher-Order Functions

I sada je potrebno da se podsetimo već spomenutog pojma funkcija višeg reda. Naime, to su funkcije koje prihvataju jednu ili više drugih funkcija kao ulazne parametre ili emituju funkciju kao povratnu vrednost. Stoga je upravo prikazana funkcija `sum()` primer jedne Higher-Order funkcije.

Singleton dizajn šablon

Samopozivajuće funkcije i Closurei osnova su za praktičnu realizaciju još jednog veoma popularnog dizajn šablona u JavaScriptu. Reč je o softverskom dizajn šablonu Singleton. Singleton je dizajn šablon koji propisuje postojanje samo jedne instance nekog tipa. Drugim rečima, Singleton nalaže rukovanje nekim tipom koje se uvek obavlja korišćenjem jedne iste instance. Tako, poštovanjem Singleton šablona, nije moguće kreirati dva objekta jednog istog tipa.

Singleton je jedan od najpopularnijih softverskih dizajn šablona koji se primenjuje kako na backendu, tako i na frontendu. U JavaScriptu Singleton se može koristiti za realizaciju različitih ponašanja koja zahtevaju postojanje samo jednog objekta tokom čitavog životnog toka stranice. Pored toga, veliki broj JavaScript biblioteka interno koristi upravo Singleton šablon kako bi ograničile kreiranje dodatnih instanci biblioteka na jednoj stranici.

U nastavku će realizacija Singleton softverskog šablona biti ilustrovana na primeru jednog objekta koji čuva podatke o korisniku, odnosno posetiocu stranice. S obzirom na to da je iz ugla frontenda posetilac uvek samo jedan, potpuno je logično da vrednosti i ponašanja u vezi sa korisnikom budu modelovana tipom koji može imati samo jednu instancu.

```
var User = (function () {

    var instance;

    function createInstance() {

        var numberOfClicks = 0;
        var dateTimeStart;
        var dateTimeEnd;

        return {

            incrementNumberOfClicks: function () {
                numberOfClicks++;
            },

            startSession: function () {
                dateTimeStart = new Date();
            },

            endSession: function () {
                dateTimeEnd = new Date();
            },

            getNumberOfClicks: function () {
                return numberOfClicks;
```

```
                    },
                    getDateTimeStart: function () {
                        return dateTimeStart.toDateString();
                    },

                    getDateTimeEnd: function () {

                        if (dateTimeEnd !== undefined) {
                            return dateTimeEnd.toDateString();
                        }

                        return "-";
                    }

                };

            };

            return {

                getInstance: function () {

                    if (!instance) {
                        instance = createInstance();
                    }

                    return instance;
                }

            };

        })();
```

Unutar upravo prikazane anonimne, samopozivajuće funkcije realizovan je Singleton dizajn šablon. Odmah možete primetiti sličnost sa nešto ranije prikazanom realizacijom Module šablona. Naime, Module šablon je osnova za realizaciju brojnih drugih šablona u JavaScriptu, pa je tako i sa Singletonom.

Unutar prikazane anonimne funkcije, na početku potrebno je uvideti tri osnovne celine:

- promenljiva **instance** - koristi se za čuvanje reference na instancu Singleton objekta;
- funkcija **createInstance()** - koristi se za kreiranje Singleton objekta;
- objekat koji se emituje kao povratna vrednost.

Izvršavanjem prikazane anonimne funkcije, promenljivoj `User` dodeljuje se objekat, koji se iz funkcije emituje kao povratna vrednost. Takav objekat poseduje samo jednu metodu – **getInstance()**. Reč je o metodi kojom se dobija Singleton objekat. Unutar nje se prvo proverava da li je objekat prethodno kreiran. Ukoliko nije, kreira se pozivanjem metode `createInstance()`. Ukoliko je objekat prethodno već kreiran, ne obavlja se njegovo ponovno kreiranje, već se isporučuje referenca koja je prethodno smeštena unutar

promenljive `instance`. Upravo opisani koraci osnovni su mehanizam koji osigurava da će svaki poziv funkcije `getInstance()` uvek rezultovati isporukom identičnog objekta.

Objekat koji se dobija pozivanjem metode `getInstance()` emituje se kao povratna vrednost ove metode. Možete videti da je reč o klasičnom Closureu. Funkcija `getInstance()` poseduje tri promenljive (`numberOfClicks`, `dateTimeStart`, `dateTimeEnd`), koje su privatne. Njima se ne može pristupiti izvan kreirane funkcije. Stoga se unutar Singleton objekta nalazi nekoliko metoda kojima se omogućava interakcija sa ovim svojstvima. Ideja je da se unutar Singleton objekta čuvaju neke osnovne informacije o korisničkoj sesiji – kada je započela, kada je završena i koliko je puta korisnik kliknuo mišem negde unutar stranice. Sesija se kreira pozivanjem metode `startSession()` i tom prilikom beleže se datum i vreme kada je tako nešto obavljeno. Završetak sesije se obavlja pozivanjem metode `endSession()` i tom prilikom se takođe obavlja smeštanje datuma u odgovarajuću privatnu promenljivu. Singleton objekat poseduje i metodu `incrementNumberOfClicks()` kojom je moguće upisati novi korisnički klik, ali i tri metode za čitanje vrednosti privatnih promenljivih.

Singleton objekat se može dobiti na sledeći način:

```
var user = User.getInstance();
```

Bez obzira na to koliko puta mi pozvali metodu `getInstance()`, ona će uvek isporučiti identičan objekat:

```
var user = User.getInstance();
var user2 = User.getInstance();

console.log(user === user2);
```

Primer rezultuje ispisom vrednosti `true` unutar konzole, što je dokaz da promenljive `user` i `user2` poseduju reference na identičan objekat.

Kod kojim se kreirani Singleton može praktično iskoristiti na nekoj stranici izgleda ovako:

```
var user = User.getInstance();
user.startSession();

document.addEventListener("click", function () {
    user.incrementNumberOfClicks();
});


var statusButton = document.getElementById("show-status-
button");

statusButton.addEventListener("click", function () {
    alert("Session start: " + user.getDateTimeStart() + "\n" +
        "Session end: " + user.getDateTimeEnd() + "\n" +
        "Number of clicks: " + user.getNumberOfClicks());
});

window.onbeforeunload = function () {

    user.endSession();
    //save state if you like
```

```
        };
```

U primeru se prvo dolazi do Singleton objekta. Zatim se poziva metoda za započinjanje sesije. Nakon toga je obavljena registracija logike koja će se aktivirati prilikom svakog klika unutar stranice. Tom prilikom pozivaće se `incrementNumberOfClicks()` metoda Singleton objekta, te ćemo na ovaj način moći da pratimo broj klikova korisnika.

Primer podrazumeva i postojanje jednog `button` elementa unutar HTML koda. Zbog toga primer poseduje i kod koji obrađuje klik na takav `button` element. Klikom na `button` element prikazuje se status korisnika – vreme početka i kraja sesije i broj klikova.

Na kraju, poslednjim blokom koda obavljena je pretplata i na događaj koji se aktivira neposredno pre nego što se stranica napusti. Tada se poziva metoda za zatvaranje sesije. Dodat je i komentar, na čije mesto je moguće postaviti kod za čuvanje prikupljenih podataka o korisniku njihovim slanjem, na primer, nekom udaljenom serveru.

Pitanje

Singleton dizajn šablon podrazumeva:

- **postojanje samo jedne instance nekog tipa;**
- postojanje maksimalno dve instance nekog tipa;
- postojanje samo jedne klase sa nekim nazivom;
- postojanje maksimalno dve klase sa istim nazivom.

Objašnjenje:

Singleton je dizajn šablon koji propisuje postojanje samo jedne instance nekog tipa. Drugim rečima, Singleton nalaže rukovanje nekim tipom koje se uvek obavlja korišćenjem jedne iste instance.

Rezime

- prilikom kreiranja JavaScript funkcije, ona postaje jedna oblast važenja, s obzirom na to da je reč o bloku koda;
- kada se promenljiva deklariše izvan funkcije, takva promenljiva postaje globalna promenljiva;
- kada se promenljiva deklariše unutar jedne funkcije, ona je vidljiva samo unutar takve funkcije, a drugačije se naziva lokalna promenljiva;
- ukoliko se unutar neke funkcije inicijalizuje promenljiva koja prethodno nigde nije deklarisana, nastaje implicitno globalna promenljiva;
- samopozivajuće funkcije se aktiviraju automatski onoga trenutka kada izvršavanje koda dođe do njih, bez potrebe da budu eksplicitno pozvane;
- JavaScript omogućava smeštanje funkcije unutar funkcije;
- Closure je pojam koji se odnosi na spoj funkcije i pripadajuće leksičke oblasti unutar koje je takva funkcija definisana;
- JavaScript funkcije mogu emitovati funkciju kao svoju povratnu vrednost;
- funkcija koja kao svoju povratnu vrednost emituje funkciju naziva se Higher-Order funkcija;
- Singleton je dizajn šablon koji propisuje postojanje samo jedne instance nekog tipa.