Tajming i Callback funkcije

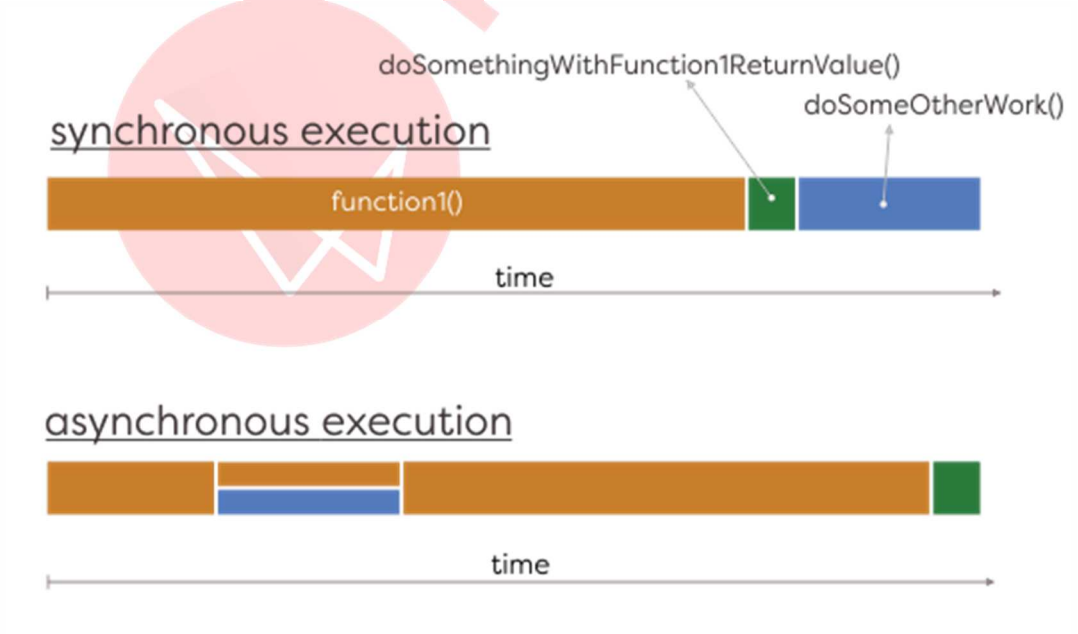
U prethodnoj lekciji ilustrovani su osnovni postulati višenitnog programiranja u JavaScriptu. Mogli ste da pročitate da se JavaScript programski kod primarno izvršava unutar jedne niti. Dodatne, radne niti je moguće kreirati korišćenjem Web Workers API-ja. Ipak, dodatne niti koje se na ovaj način dobijaju mogu se koristiti samo u ograničenom broju slučajeva – takav kod mora biti u zasebnom fajlu i može se koristiti samo za obavljanje proračuna i kalkulacija. Drugim rečima, niti koje se dobijaju upotrebom Web Workers API-ja ne mogu da pristupe globalnom `Window` objektu, pa samim tim i nisu u mogućnosti da vrše manipulaciju nad DOM strukturom elemenata.

Ukoliko se izuzme Web Workers API, koji se može koristiti samo u nekim situacijama, može se reći da JavaScript obezbeđuje jednonitni (*engl. single-threaded*) model izvršavanja. Ipak, kako bi se sprečilo blokiranje takve jedne niti izvršavanja, web pregledači poseduju određene funkcionalnosti koje je moguće koristiti kako bi se postiglo izvršavanje većeg broja operacija tokom istog vremenskog perioda. Takve tehnike omogućavaju da se pored podrazumevanog linijskog, odnosno *sinhronog* modela, u JavaScriptu postigne i *asinhrono* izvršavanje. O takvom modelu izvršavanja programskog koda u jeziku JavaScript biće reči u nastavku ovoga modula.

Šta je sinhrono, a šta asinhrono izvršavanje?

Na samom početku, potrebno je definisati nekoliko veoma važnih pojmova kako biste na pravi način mogli da razumete praktičnu materiju koja će biti izneta u redovima ove i naredne lekcije. Upravo u uvodnom izlaganju spomenuti su pojmovi *sinhrono* i *asinhrono*. O čemu je reč i koja su značenja ovih pojmova?

Izvršavanje nekog programskog koda, bez obzira na jezik koji se koristi, može biti sinhrono ili asinhrono. Osnovne razlike između ova dva načina izvršavanja programskog koda ilustrovane su slikom 8.1.



Slika 8.1. Uporedni prikaz sinhronog i asinhronog izvršavanja

Slika 8.1. ilustruje izvršavanje tri JavaScript funkcije:

- `function1()`
- `doSomethingWithFunction1ReturnValue()`
- `doSomeOtherWork()`

Nazivi funkcija su definisani tako da oslikavaju osnovnu namenu svake od njih. Funkcija `function1()` obavlja vremenski najdužu operaciju. Kada svoju radnju završi, isporučuje povratnu vrednost koja se koristi od strane funkcije `doSomethingWithFunction1ReturnValue()`. Na kraju, funkcija `doSomeOtherWork()` je nezavisna od dve prethodne funkcije.

Gornja polovina slike 8.1. ilustruje primer sinhronog izvršavanja programskog koda. Sinhrono izvršavanje podrazumeva izvršavanje samo jedne funkcije u jednom trenutku. Drugim rečima, kako bi započelo izvršavanje sledeće funkcije, prethodna prvo mora završiti svoje izvršavanje. Takvo ponašanje je potpuno korektno kada su u pitanju funkcije `function1()` i `doSomethingWithFunction1ReturnValue()`. Naime, funkcija `doSomethingWithFunction1ReturnValue()` svakako mora da čeka na završetak funkcije `function1()`. Ipak, to nije slučaj i sa funkcijom `doSomeOtherWork()`. S obzirom na to da ne zavisi od preostale dve funkcije, ona nije u obavezi da čeka njihov završetak. Ipak, sinhrono izvršavanje podrazumeva obavljanje samo jedne operacije u jednom trenutku, pa je u takvom modelu izvršavanja na neki način funkcija `doSomeOtherWork()` blokirana, odnosno dužna je da čeka završetak funkcija koje su pozvane pre nje.

Kod asinhronog izvršavanja situacija je nešto drugačija. Više različitih operacija se mogu izvršavati tokom istog vremenskog perioda. To ilustruje donja polovina slike 8.1. Možete videti da na početku započinje izvršavanje funkcije `function1()`. Ipak, dok funkcija `function1()` obavlja svoj posao, izvršava se i funkcija `doSomeOtherWork()`, s obzirom na to da ona nije u obavezi da čeka završetak funkcije `function1()`. Onoga trenutka kada i `function1()` završi svoj posao, obaveštava se i `doSomethingWithFunction1ReturnValue()` funkcija, koja povratnu vrednost `function1()` funkcije koristi za svoje izvršavanje.

Sa slike 8.1. može se jasno videti osnovna prednost asinhronog izvršavanja – ono omogućava da se više operacija obavlja tokom istog vremenskog perioda. Drugim rečima, asinhrono izvršavanje omogućava da se jedna funkcija pozove, a kada ona završi svoj posao, da se dobije dojava o završetku i eventualna povratna vrednost. U međuvremenu se mogu izvršavati neke druge operacije, koje nisu zavisne od one na čiji rezultat se čeka.

Pojmovi sinhronog i asinhronog izvršavanja za vas u ovom trenutku mogu biti zbunjujući. Naravno, uvek je najbolje nove pojmove i osobine jezika upoznati na praktičnim primerima. Stoga će u nastavku biti prikazani realni primeri sinhronog i asinhronog izvršavanja u JavaScriptu.

Sinhrono, asinhrono i višenitno izvršavanje

Bitno je razumeti da pojmovi asinhronog i višenitnog programiranja nisu sinonimi. U prethodnoj lekciji prikazani su osnovni postulati višenitnog izvršavanja. Tako ste mogli da pročitate da **višenitno izvršavanje** podrazumeva samo jedno – **izvršavanje koda u više zasebnih niti.**

Sa druge strane, **asinhrono izvršavanje** podrazumevano znači samo sledeće – **kod se ne izvršava linearno, te je moguće neku funkciju pozvati i pritom ne čekati eksplicitno na njen završetak.**

Asinhrono izvršavanje se može realizovati korišćenjem više niti, ali tako nešto nije obavezno. Drugim rečima, asinhrono izvršavanje je moguće postići i unutar samo jedne niti. U takvim slučajevima, izvršavanje većeg broja operacija tokom istog vremenskog perioda postiže se tako što se izvršavanje većeg broja operacija obavlja ciklično, pri čemu svaka od operacija tokom jednog ciklusa dobija određeni, vrlo kratak vremenski period za izvršavanje. Brzim smenjivanjem izvršavanja više operacija, korisnik ima privid da se one izvršavaju u istom trenutku.

Takođe, ni višenitno programiranje ne mora uvek podrazumevati asinhrono izvršavanje. Upravo je takva situacija bila ilustrovana i u prethodnoj lekciji. Naime, kod unutar svih radnih niti koje su dobijene korišćenjem Web Workers API-ja izvršava se sinhrono. Stoga, iako Web Workers API omogućava kreiranje dodatnih niti, kod unutar svake od njih izvršava se sinhrono.

Pitanje

Ukoliko se kod izvršava naredbu po naredbu, pri čemu svaka naredba mora čekati završetak prethodne, reč je o:

- **sinhronom izvršavanju;**
- asinhronom izvršavanju;
- asocijalnom izvršavanju;
- stacionarnom izvršavanju.

Objašnjenje:

Sinhrono izvršavanje podrazumeva izvršavanje više operacija linearno, odnosno izvršavanje jedne operacije za drugom.

Primer sinhronog izvršavanja u JavaScriptu

Primer sinhronog izvršavanja JavaScript koda biće kreiran korišćenjem dobro poznate metode `alert()` `Window` objekta:

```
<button id="my-button">Click me</button>

<script>
    var myButton = document.getElementById("my-button");

    myButton.addEventListener("click", function() {

        alert("Hello from modal window");

        console.log("Hello from statement after alert() method.");

    });
</script>
```

Primer podrazumeva jedan HTML element (`button`) i blok JavaScript koda kojim se vrši pretplata na `click` događaj na takav `button` element. Klik na `button` element aktivira dve naredbe. Prva naredba podrazumeva pozivanje metode `alert()` za prikaz poruke korisniku. Druga naredba podrazumeva pozivanje metode `log()`, `console` objekta, za ispis poruke unutar konzole web pregledača. Ipak, ono što je bitno primetiti jeste da se nakon klika na `button` element neće izvršiti obe definisane naredbe. Naime, klikom na `button` element, prvo će se izvršiti naredba kojom se poziva `alert()` metoda. Bitno je da znate da je ovakva metoda aktivna sve dok je modalni prozor prikazan. Tek kada korisnik klikne na *OK* dugme unutar modalnog prozora koji se dobija metodom `alert()`, prelazi se na sledeću naredbu. Ovo je klasičan primer sinhronog izvršavanja JavaScript koda.

Može se reći da je metoda `alert()` sinhrona metoda, jer u potpunosti blokira izvršavanje JavaScripta sve dok je ona aktivna, a aktivna je sve dok korisnik ne klikne na OK dugme unutar modalnog prozora.

Primer asinhronog izvršavanja u JavaScriptu

Pored sinhronih funkcija, web pregledači u skupu funkcionalnosti, koje objedinjene pod `Window` objektom izlažu našem JavaScript kodu, poseduju i one koje se izvršavaju asinhrono, odnosno izvršavaju se redosledom koji se razlikuje od onog koji je definisan u kodu. Zapravo, upravo prikazani primer, koji je oslikavao sinhrono izvršavanje, ujedno ilustruje i asinhrono:

```
<button id="my-button">Click me</button>

<script>
    var myButton = document.getElementById("my-button");

    myButton.addEventListener("click", function () {

        alert("Hello from modal window");

        console.log("Hello from statement after alert() method.");

    });

    console.log("Hello from some other code.");

</script>
```

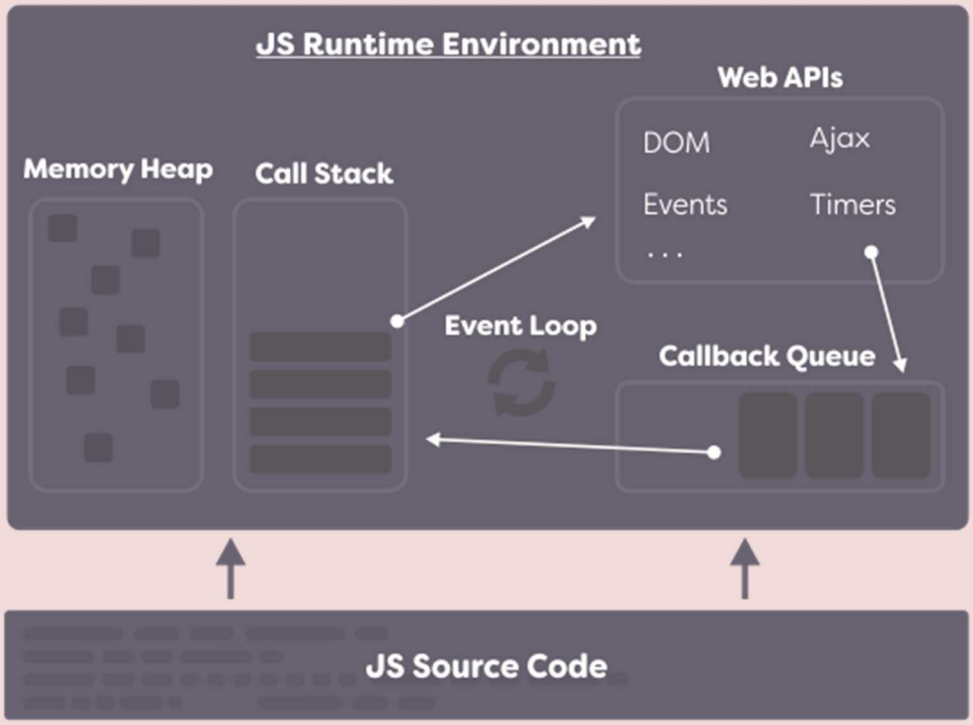
Bitno je da primetite da se u primeru poziva jedna funkcija, čijom upotrebom se utiče na redosled izvršavanja definisanih operacija. Naravno, reč je o funkciji `addEventListener()`, kojom se u prikazanom primeru obavlja definisanje logike koja će se izvršiti prilikom klika na `button` element. Drugim rečima, sada se ne čeka da korisnik zaista i klikne na `button` element kako bi se nastavilo sa obavljanjem narednih operacija, kao što je to bio slučaj u prethodnom primeru prilikom upotrebe metode `alert()`. Ovakvo ponašanje je moguće, zato što funkcija `addEventListener()` kao jedan od svojih parametara prihvata i referencu na funkciju, koju je potrebno izvršiti kada dođe do događaja koji se sluša. Tako se na ovaj način dobija situacija kao na slici 8.1. iz ove lekcije – prvo se obavlja registrovanje logike koja će se izvršiti u nekom trenutku u budućnosti. Nakon toga, nastavlja se normalno izvršavanje logike koja se nalazi u nastavku. Na kraju, kada korisnik klikne na `button` element, aktivira se i anonimna funkcija koja je metodi `addEventListener()` prosleđena kao parametar.

Kako funkcioniše model izvršavanja koda u JavaScriptu?

Već je rečeno da se JavaScript kod koji mi pišemo izvršava unutar jedne niti, ali da može da iskoristi blagodeti asinhronog modela izvršavanja. Ipak, iako se JavaScript izvršava unutar samo jedne niti, to ne znači da i JavaScript izvršno okruženje koristi samo jednu nit. Ono za izvršavanje različitih ugrađenih funkcionalnosti web pregledača može koristiti, a veoma često i koristi, veći broj niti.

Kako bi sve ovo postalo jasnije, u nastavku će biti objašnjen osnovni princip izvršavanja JavaScript koda koji koristi neku asinhronu operaciju.

Slika 8.2. ilustruje osnovne komponente koje učestvuju u izvršavanju JavaScript koda.



Slika 8.2. Način na koji se izvršava naredba kojom se poziva neka asinhrona operacija web pregledača

Izvršavanje JavaScript koda podrazumeva nekoliko različitih komponenata koje su objedinjene pod pojmom JavaScript izvršnog okruženja (*engl. JS Runtime Environment*). JavaScript izvršno okruženje sastavni je deo svakog web pregledača. Ipak, web pregledači nisu jedini programi koji mogu da izvršavaju JS kod. Tako se JavaScript izvršno okruženje nalazi i unutar programa kao što je Node.js.

Početak izvršavanja JavaScript koda koji mi samostalno pišemo podrazumeva njegovo konvertovanje u mašinski oblik. Unutar hipa (**Memory Heap**) smeštaju se sve vrednosti, odnosno promenljive, a unutar steka poziva (**Call Stack**) pozivi svih funkcija koji su definisani u kodu. Bitno je da primetite da funkcionalnosti koje omogućavaju asinhrono izvršavanje nisu sastavni deo JavaScripta. One su zasebni deo izvršnog okruženja (**Web APIs**), koje web pregledači izlažu na korišćenje kodu koji mi samostalno pišemo. Pri tome, nikako ne treba shvatiti da su sve funkcionalnosti koje web pregledači nama izlažu na korišćenje asinhrone. Postoji veliki broj funkcionalnosti različitih Web API-ja koje se izvršavaju sinhrono (npr. alert() metoda), dok su one za koje postoji realna potreba realizovane direktno od strane web pregledača kao asinhrone. Neke od njih su funkcije za pretplatu na događaje, slanje HTTP zahteva, tajming funkcije...

Kada unutar steka poziva red izvršavanja dođe do neke asinhrone funkcije web pregledača, on se uklanja iz steka poziva i prebacuje odgovarajućem Web API-ju. Prilikom poziva takvih asinhronih funkcija, prilaže se i callback funkcija, čija je svrha dojava o završetku asinhrone operacije. Stoga, kada neka od asinhronih operacija web pregledača završi svoj posao, ona prosleđenu callback funkciju prebacuje u jednu posebnu strukturu – red callback funkcija (**Callback Queue**). Funkcija iz takvog reda šalje se na izvršavanje unutar steka poziva tek onda kada je takav stek u potpunosti ispražnjen, kako ne bi došlo do blokiranja operacija koje unutar njega čekaju da budu izvršene.

Ovo je bio osnovni princip funkcionisanja asinhronog modela u JavaScriptu. Postojanje reda callback funkcija i razdvojenost Web API-ja od samog jezika omogućavaju da se operacije ne izvršavaju onim redom kojim su definisane, te da se na taj način spreči blokiranje niti unutar koje se izvršava naš JavaScript kod.

Slika 8.2. ilustruje i ono što je već rečeno: iako se kod koji mi samostalno pišemo izvršava unutar jedne niti, to ne mora biti slučaj i sa funkcionalnostima koje web pregledači nama izlažu na korišćenje (Web APIs). Naime, način na koji su realizovane različite funkcionalnosti Web API-ja zavisi isključivo od web pregledača, a u dosta slučajeva za izvršavanje velikog broja takvih operacija web pregledači interno koriste dodatne niti.

Na slici 8.2. postoji još jedan pojam koji nije objašnjen – **Event Loop**. Uprošćeno rečeno, reč je o jednoj petlji koja konstantno proverava Callback Queue i Call Stack. Tako Event Loop redom šalje na izvršavanje sve pozive funkcija koje se nalaze unutar steka poziva. Kada se stek poziva isprazni, identično se obavlja i sa callback funkcijama, koje na svoje izvršavanje čekaju unutar reda callback funkcija.

Tajming funkcije

Jedan od načina na koji se u JavaScriptu može uticati na redosled izvršavanja operacija jeste korišćenje tajming funkcija (*engl. timing function*):

- `setTimeout()` – izvršava neku funkcionalnost jednom, nakon što protekne određena količina vremena;
- `setInterval()` – izvršava određenu funkcionalnost iznova i iznova, sa određenim fiksnim razmacima između svakog izvršavanja.

Obe prikazane funkcije koriste se na identičan način. Njima se kao parametar prosleđuje referenca na funkciju koju je potrebno izvršiti jednom nakon nekog vremena ili više puta uz određeni vremenski razmak.

setTimeout()

Prvo će biti ilustrovane osobine `setTimeout()` funkcije. Reč je o funkciji koja omogućava da se određeni kod izvrši jedanput u budućnosti, nakon određenog vremenskog perioda. Sintaksa ove funkcije je sledeća:

```
setTimeout(function, delay, arg);
```

Može se videti da funkcija `setTimout()` može da prihvati sledeće parametre:

- `func` – funkcija koja će se izvršiti nakon nekog vremenskog perioda; ovo je obavezan parametar;
- `delay` – vremenski period nakon koga će prosleđena funkcija biti aktivirana; vrednost se izražava u milisekundama; ovaj parametar nije obavezan i, ukoliko se ne prosledi, za njegovu vrednost se uzima 0;
- `arg` – proizvoljan broj parametara koji će biti prosleđeni `func` funkciji.

Nakon upoznavanja osnovnih osobina funkcije `setTimeout()`, na redu je upoznavanje i sa njenim ponašanjem na praktičnim primerima:

```
console.log("Statement 1");

setTimeout(function(){
    console.log("Statement 2");
}, 2000);

console.log("Statement 3");
```

Definisani kod poseduje tri naredbe. Sve one obavljaju štampanje poruka unutar konzole. Tekst koji ispisuju unutar konzole slikovit je i pokušava da dočara redosled kojim su ove tri naredbe definisane unutar JavaScript koda (1, 2, 3). Ipak, kada se ovakav kod pokrene, unutar konzole se dobija sledeći ispis:

```
Statement 1
Statement 3
Statement 2
```

Na osnovu ispisa u konzoli može se zaključiti da je prvo izvršena prva naredba, onda treća, a na kraju i druga. Razlog je i više nego očigledan. Druga naredba (*Statement 2*) smeštena je unutar anonimne funkcije koja je prosleđena metodi `setTimeout()`. Pritom je definisano da će se prosleđena funkcija izvršiti nakon 2.000 milisekundi (2 sekunde). S obzirom na to da je `setTimeout()` jedna od metoda web pregledača koje se izvršavaju asinhrono, ne dolazi do blokiranja koda JavaScript jezika, pa se tokom čekanja na izvršavanje prosleđene anonimne funkcije ostatak logike nesmetano izvršava.

Zanimljiv efekat se dobija u sledećoj situaciji:

```
console.log("Statement 1");

setTimeout(function(){
    console.log("Statement 2");
}, 0);

console.log("Statement 3");
```

Kod je identičan prethodnom primeru, uz jednu malu razliku – za vreme nakon koga je potrebno pokrenuti funkciju koja se prosleđuje `setTimeout()` metodi definisano je 0 milisekundi. To praktično znači da je prosleđenu funkciju potrebno pokrenuti odmah, odnosno bez zadrške. Ipak, tako nešto se ne događa:

```
Statement 1
Statement 3
Statement 2
```

Unutar konzole se dobija identičan ispis, iz čega se može zaključiti da je redosled izvršavanja naredbi identičan (1, 3, pa 2).

Zbog čega se dobija ovakvo ponašanje možete pročitati u prethodnom poglavlju: *Kako funkcioniše model izvršavanja koda u JavaScriptu?* Ukratko, bez obzira na vreme koje se navede kao parametar, funkcija koja se prosledi metodi `setTimeout()` uvek se izvršava tek kada se u potpunosti oslobodi stek poziva. U prikazanom primeru, to je razlog zbog koga se prvo izvršava treća naredba. Njenim izvršavanjem oslobađa se stek poziva, pa se prelazi i na izvršavanje funkcije prosleđene metodi `setTimeout()`.

Vreme definisano parametrom delay nije zagarantovano

Iz prethodnih redova se može naslutiti još jedna veoma važna osobina `setTimeout()` metode. Vreme koje se definiše parametrom `delay` nije zagarantovano. Iako smo definisali vrednost 0, funkcija se ne izvršava odmah, već tek onda kada se stek poziva oslobodi. Identično se može dogoditi i kada se definiše neka druga vrednost – na primer, 1.000 milisekundi. Bez obzira na duži vremenski period, ukoliko se nakon poziva `setTimeout()` funkcije nalazi neka vremenski zahtevna operacija, period koji će proći do početka izvršavanja funkcije može biti i duži. Ipak, bitno je znati da period nikada ne može biti kraći od definisanog.

Otkazivanje timeouta

Otkazivanje logike koja se prosleđuje `setTimeout()` metodi može se obaviti pozivanjem još jedne specijalne metode **clearTimeout()**. Ovakva metoda prihvata jedan parametar koji se odnosi na identifikator prethodno pozvane `setTimeout()` metode:

```
var timeout1 = setTimeout(function () {
    console.log("Statement 2");
}, 2000);

clearTimeout(timeout1);
```

Pozivanjem metode `setTimeout()` ona emituje povratnu vrednost koja se odnosi na identifikator takvog timeouta. Zbog toga, u prikazanom primeru takva povratna vrednost uhvaćena je u jednu promenljivu, a zatim i iskorišćena kao ulazni parametar metode `clearTimeout()`. Stoga se u navedenom primeru anonimna metoda koja štampa vrednost *Statement 2* uopšte neće ni izvršiti, s obzirom na to da se odmah nakon pozivanja `setTimeout()` metode poziva i metoda `clearTimeout()`.

Prosleđivanje parametara setTimeout() metodi

U nekim situacijama poželećete da funkciji koju prosleđujete `setTimeout()` metodi prosledite i neki dodatni parametar. To se može postići na sledeći način:

```
var timeout1 = setTimeout(function (name) {
    console.log("Hello " + name);
}, 2000, "Ben");
```

Parametar koji je potrebno proslediti funkciji koju će aktivirati `setTimeout()` metoda definiše se kao treći parametar. U primeru je to vrednost *Ben*. Dalje, anonimna funkcija koja se prosleđuje metodi `setTimeout()` sada poseduje i jedan ulazni parametar. On će prilikom pozivanja da dobije vrednost *Ben*. Stoga, nakon dve sekunde, unutar konzole ispisuje se:

```
Hello Ben
```

Primer – otkazivanje jednog timeouta od strane drugog

Za kraj priče o timeoutima, biće prikazan i jedan primer koji objedinjuje prosleđivanje parametara i otkazivanje timeouta.

```
console.log("Statement 1");

var timeout1 = setTimeout(function () {
    console.log("Statement 2");
}, 2000);

var timeout2 = setTimeout(function (timeout) {
    clearTimeout(timeout);
    console.log("Timeout 1 is cleared...");
}, 1000, timeout1);

console.log("Statement 3");
```

Primer sada poseduje dva poziva `setTimeout()` metode. Prvi poziv izgleda kao i dosad – izvršiće ispis poruke *Statement 2* unutar konzole nakon dve sekunde. Drugi poziv `setTimeout()` metode definiše funkciju kojoj se prosleđuje identifikator prvog timeouta. Takav identifikator se koristi da se nakon jedne sekunde otkaže prvi timeout. Stoga, pokretanjem prikazanog koda, unutar konzole dobija se:

```
Statement 1
Statement 3
Timeout 1 is cleared...
```

Može se videti da se logika prvog timeouta uopšte ne izvršava. Drugim rečima, ovo je primer kako jedan timeoute otkazuje drugi. S obzirom na to da se drugi timeout aktivira pre prvog, njegova logika je u mogućnosti da otkaže prvi timeout.

setInterval()

Još jedna tajming funkcija koju web pregledači izlažu na korišćenje kodu koji mi samostalno pišemo jeste i funkcija `setInterval()`. Za razliku od `setTimeout()` funkcije, ova funkcija omogućava da se logika jedne funkcije izvršava iznova i iznova, sa jednakim, unapred definisanim razmacima između izvršavanja.

Metoda `setInterval()` koristi se na identičan način kao i već prikazana `setTimeout()` metoda:

```
setInterval(func, delay, arg);
```

Može se videti da je sintaksa metode `setInterval()` identična sintaksi `setTimeout()` metode. Ona može da prihvati sledeće parametre:

- `func` – funkcija koja će se izvršavati iznova i iznova;
- `delay` – vremenski razmak između dva susedna poziva `func` funkcije; vrednost se izražava u milisekundama; u ovom slučaju reč je o obaveznom parametru; ipak većina web pregledača kao minimalan razmak između dva izvršavanja definiše vrednost od 4 ms;
- `arg` – proizvoljan broj parametara koji će biti prosleđeni `func` funkciji.

S obzirom na to da se koristi na gotovo identičan način kao i `setTimeout()`, upotreba metode `setInterval()` biće ilustrovana na samo jednom primeru:

```
        <h1 id="heading">setInterval()</h1>

        <script>

            let heading = document.getElementById("heading");

            function changeColor(elem) {
                elem.style.color = elem.style.color == 'red' ? 'blue' :
'red';
            }

            setInterval(changeColor, 500, heading);

        </script>
```

Primer definiše jedan HTML element (`h1`) sa određenim tekstom. Unutar `script` elementa prvo se dolazi do reference na takav `h1` element. Zatim se definiše funkcija koja prihvata jedan parametar (`elem`) i svakim pozivom menja boju teksta prosleđenog elementa između dve boje (red i blue). Na kraju, obavlja se pozivanje metode `setInterval()`. Njoj se prosleđuju tri parametra:

- `changeColor` – funkcija koja će se pozivati iznova i iznova;
- `500` – vreme u milisekundama između pojedinačnih poziva metode `changeColor`
- `heading` – parametar koji se prosleđuje metodi `changeColor`

Sve ovo stvoriće efekat prikazan animacijom 8.1.

Animacija 8.1. Efekat dobijen upotrebom setInterval() funkcije

Kao što vidite, primer će stvoriti efekat naizmenične promene boje teksta naslova. Takav efekat će trajati neprekidno, odnosno sve dok je stranica otvorena. Ukoliko je potrebno ograničiti trajanje kreiranog efekata, mogu se kombinovati `setInterval()` i `setTimeout()` metode:

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

            function changeColor(elem) {
                elem.style.color = elem.style.color == 'red' ? 'blue' :
'red';
            }

            let interval1 = setInterval(changeColor, 500, heading);

            setTimeout(function(intervalId){
                clearInterval(intervalId);
            }, 5000, interval1);
```

Sada je primeru dodat i poziv metode `setTimeout()`. Korišćenjem ove metode, nakon pet sekundi biće aktivirana anonimna funkcija kojom će biti otkazan aktivni interval. Otkazivanje se obavlja korišćenjem metode **clearInterval()**, kojoj se prosleđuje identifikator intervala. Tako će na ovaj način efekat naizmenične promene boje trajati samo pet sekundi.

Zanimljivost

Funkcionalnost koju JavaScript izvršno okruženje koristi za generisanje identifikatora timeouta i intervala je zajednička. Drugim rečima, ne može se dogoditi da interval i timeout poseduju identičan identifikator. Zato se metode `clearTimeout()` i `clearInterval()` mogu smatrati sinonimima. Stoga, bez obzira na to da li je reč o intervalu ili timeoutu, vi možete iskoristiti bilo koju od ove dve funkcije. Ipak, zbog preglednosti i boljeg razumevanja koda koji se piše, savetuje se korišćenje funkcije odgovarajućeg naziva.

Callback funkcije

Dosad je prikazano nekoliko različitih funkcionalnosti web pregledača, koje je moguće koristiti kako bi se definisao kod koji se neće izvršiti u tradicionalnom, linijskom, odnosno sinhronom maniru, nego tek onda kada dođe do pojave nekog događaja ili nakon određenog vremenskog perioda. Ipak, bitno je znati da takav model izvršavanja ne bi bio moguć bez upotrebe jedne posebne vrste funkcija – callback funkcija.

Svi primeri u ovoj lekciji podrazumevali su upotrebu callback funkcija. One su korišćene kako bi se definisalo šta će se dogoditi kada ugrađene, asinhrone funkcije završe svoj posao. Tako su callback funkcije prvo korišćene u primerima pretplate na događaje. Metoda `addEventListener()` kao svoj parametar prihvata upravo callback funkciju. Dalje, tajming funkcije takođe kao jedan od svojih parametara prihvataju callback funkciju. Tako se može reći da su callback funkcije one koje se kao argumenti prosleđuju metodama koje svoju logiku obavljaju u pozadini. Onoga trenutka kada se takva pozadinska logika završi, dojava o završetku se obavlja pozivanjem callback funkcije. Upravo zbog toga su callback funkcije dobile svoj naziv.

Callback funkcije se intenzivno koriste u ogromnom broju ugrađenih funkcionalnosti web pregledača, što se moglo videti i u dosadašnjem toku ove lekcije. Ipak, u poslednje vreme, pogotovu u novijim Web API-jima, callback funkcije su dobile alternativu u vidu modernije i moćnije tehnologije – Promise. O takvim novim pristupima biće reči u narednoj lekciji.

Rezime

- višenitno izvršavanje podrazumeva izvršavanje koda u više zasebnih niti;
- sinhrono izvršavanje podrazumeva izvršavanje više operacija linearno, odnosno izvršavanje jedne operacije za drugom;
- asinhrono izvršavanje podrazumeva da se kod ne izvršava linearno, te je moguće neku funkciju pozvati i pritom ne čekati eksplicitno na njen završetak;
- JavaScript podrazumevano obezbeđuje jednonitni, asinhroni model izvršavanja;
- JavaScript kod se izvršava unutar komponente web pregledača koja se naziva JavaScript izvršno okruženje;
- JavaScript izvršno okruženje sastoji se iz nekoliko delova: hipa, steka poziva, reda callbak funkcija, petlje događaja i Web API-ja;
- iako se JavaScript izvršava unutar samo jedne niti, JavaScript izvršno okruženje može da koristi veći broj niti;
- jedan od načina na koji se u JavaScriptu može uticati na redosled izvršavanja operacija jeste korišćenje tajming funkcija;
- `setTimeout()` – izvršava određenu funkcionalnost jednom, nakon što protekne određena količina vremena;
- `setInterval()` – izvršava određenu funkcionalnost iznova i iznova, sa određenim fiksnim razmacima između svakog izvršavanja;
- timeoute i intervale je moguće otkazati metodama `clearTimeout()` i `clearInterval()`