

JavaScript funkcije ključni su dio jezika, ali nisu baš ono što se čini da jesu. Izgledaju kao da bi trebale pripadati obitelji iskaza, ali su zapravo objekti baš kao i svi ostali koje smo obradili u prethodnom poglavlju. Funkciju možete definirati, izraditi novu i čak ju ispisati.

Zahvaljujući toj funkcionalnosti funkciju možete dodijeliti varijabli ili elementu polja ili ju čak prosljediti kao argument pozivu druge funkcije. To funkciju čini vrlo fleksibilnim i korisnim objektom, ali i zbunjujućim za nove JavaScript programere.

Izradi funkcija u JavaScriptu može se pristupiti na tri načina: deklarativni/statički, dinamički/anonimni i doslovno. Važno je razumjeti učinak svakog tipa deklaracije prije nego što ga upotrijebite.

## Deklarativne funkcije

Najčešći tip funkcije koristi deklarativno/statički format. Ovaj pristup započinje ključnom riječi `function` koju slijedi ime funkcije, zagrade koje sadrže nula ili više argumenata i zatim tijelo funkcije:

```
function ime-funkcije (parametar1, parametar2, ..., parametarn) {  
    iskazi funkcije  
}
```

Vitičaste zagrade oko tijela funkcije su obavezne, čak i ako sadrži samo jedan iskaz.

Osim ako ne pravim novu biblioteku s novim objektima ili definiram funkcije prema potrebi, na temelju događaja, ovo je sintaksa koju najčešće koristim i dosad sam ju koristila za sve primjere u ovoj knjizi.

Deklarativna funkcija se parsira jednom, kad se stranica učita, a parsirani rezultat se koristi svaki put kad se funkcija pozove. Definiciju funkcije u kodu lako je uočiti, jednostavno ju je čitati i razumjeti te ima ograničen broj negativnih posljedica, poput

curenja memorije. Također je prihvatljivija i jednostavnija programerima koji su radili s drugim programskim jezicima.

Sljedeći odlomak koda definira funkciju u ovom formatu, koja se poziva odmah nakon što je deklarirana:

```
function sayHi(toWhom) {  
    alert("Hi " + toWhom);  
}  
sayHi("World!");
```

U prethodnom kodu pozivanje funkcije rezultira otvaranjem dijaloškog okvira koji prikazuje poruku „Hi World!“. Ako izuzmemo JavaScript pogreške, bez obzira na to koji je niz znakova proslijeđen funkciji ili koliko je puta pozvana, uvijek se koristi isti objekt funkcije i rezultat je uvijek isti: otvara se dijaloški okvir s porukom.

## Pravila imenovanja funkcija i veličina

Funkcije obavljaju posao. Prema tome, u ime funkcije dobro je uključiti glagol koji ukratko i na najbolji način opisuje njeno djelovanje. Sljedeće su primjeri dobrih imena funkcija:

- pokreceRacun
- ispisujeDatum
- obradjujeIme
- zbrajaBrojeve

Ime funkcije obično započinje glagolom iza kojeg slijedi jedna ili više imenica koje počinju velikim slovom. Ovo pravilo imenovanja nije obavezno u JavaScriptu te nije uvijek pristup koji programeri preferiraju. Njegova prednost je što se lako čita i donekle opisuje funkciju.

Ako vam je teško na ovaj način imenovati funkciju jer obavlja više poslova, mogli biste razmisliti da ju podijelite na manje cjeline, što olakšava njenu ponovnu upotrebu. Umjesto funkcije koja parsira obrazac i uzima vrijednosti, provjerava njihovu ispravnost i zatim ih koristi u Ajax pozivu upućenom poslužitelju, mogli biste izraditi funkciju koja provjerava upisane adrese elektroničke pošte ili poštanske brojeve i zatim njene rezultate koristiti u drugim funkcijama. Prednost smještanja funkcija za provjeravanje unosa u zasebne pakete je u tome što ih tada možete ponovno koristiti u drugim JavaScript aplikacijama koje budete razvijali.

Ustvari, trebalo bi biti pravilo da funkcije budu kratke, svojstvene zadaći i općenite koliko god je to moguće.

## Rezultati funkcija i argumenti

Funkcije komuniciraju s programom koji ih poziva putem argumenata koji su im proslijeđeni i vrijednosti koje vraćaju.

Varijable temeljene na osnovnim tipovima, kao što su niz znakova, logička vrijednost ili broj, prosljeđuju se funkciji prema vrijednosti. To znači da, ako promijenite argument u funkciji, promjena se neće odraziti na program koji tu funkciju poziva.

Objekti prosljeđeni funkciji, s druge strane, prosljeđuju se prema referenci. Promjene izvedene na objektu u funkciji odražavaju se na program koji poziva.

U primjeru 5-1 dva su argumenta prosljeđena funkciji: jedan je varijabla niza znakova a drugi je polje. Kôd mijenja oba argumenta u funkciji i zatim prikazuje njihov sadržaj u programu koji poziva. Niz znakova se ne mijenja jer je vrijednost osnovnog tipa, ali drugi član objekta polja ima drugačiju vrijednost te je dodan treći član polja.

*Primjer 5-1. Argumenti funkcije prosljeđeni prema vrijednost i prema referenci.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Pass Me</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//

function alterArgs(strLiteral, aryObject) {

    // Prepisuje originalni niz
    strLiteral = "Override";
    aryObject[1] = "2";
    aryObject[aryObject.length] = "three";
}

function testParams() {
    var str = "Original Literal";
    var ary = new Array("one", "two");

    document.writeln("string literal is " + str + "&lt;br /&gt;");
    document.writeln("Array object is " + ary + "&lt;br /&gt;&lt;br /&gt;");

    alterArgs(str,ary);

    document.writeln("string literal is " + str + "&lt;br /&gt; ");
    document.writeln("Array object is " + ary);
}

//]]&gt;
&lt;/script&gt;
&lt;/head&gt;
&lt;body onload="testParams();"&gt;
&lt;p&gt;Some page content&lt;/p&gt;
&lt;/body&gt;
&lt;/html&gt;</pre></div><div data-bbox="679 931 883 949" data-label="Page-Footer"><hr/><p>Poglavlje 5: Funkcije | 105</p></div>
```

Slika 5-1 prikazuje aplikaciju iz primjera 5-1 učitano u Firefox. Prva dva reda su doslovna vrijednost niza znakova i objekt polja prije poziva funkcije, a druga dva su iste vrijednosti nakon poziva funkcije.



Slika 5-1. Rezultat izvođenja aplikacije iz primjera 5-1 u Firefoxu.

Funkcija može i ne mora vratiti vrijednost. Ako ju vrati, iskaz `return` može se pojaviti bilo gdje u kodu funkcije, a moglo bi biti čak i više iskaza `return`. Kad aplikacija koja obrađuje JavaScript naiđe na iskaz `return`, ona u tom trenutku prestaje obrađivati funkciju i vraća kontrolu iskazu koji poziva.

Jedan od razloga zašto biste mogli imati više iskaza `return` je slučaj da želite završiti i napustiti funkciju kad je uvjet zadovoljen. U sljedećem odlomku koda, ako u funkciji nije udovoljeno uvjetu, funkcija odmah prestaje, a u suprotnom se obrada nastavlja:

```
function testValues(numValue) {  
    if (isNaN(numValue)) {  
        return "error -- not a number";  
    }  
    ...  
    return ...}  
}
```

Funkcije ne zahtijevaju povratne vrijednosti, iako one mogu biti korisne pri obradi pogrešaka – primjerice, vraćanje vrijednosti `false` ako funkcija nije uspješna. (Složene metode obrađivanja pogrešaka opisujem u poglavlju 13.)

Suprotna po ponašanju deklarativnoj funkciji je dinamička/anonimna funkcija koju ću opisati u sljedećem odjeljku.

# Anonimne funkcije

Funkcije su objekti. Prema tome, možete ih izraditi – baš kao String, Array ili drugi tip objekta – korištenjem konstruktora i dodjeljivanjem funkcije varijabli. U sljedećem kodu izrađena je nova funkcija pomoću konstruktora Function kojem su kao argumenti proslijeđeni tijelo i argumenti funkcije:

```
var sayHi = new Function("toWhom","alert('Hi ' + toWhom);");
sayHi("World!");
```

Ovaj tip funkcije često se naziva *anonimnom funkcijom* jer ona sama nije izravno deklarirana ili imenovana.

Aplikacija koja obrađuje JavaScript, za razliku od postupka s deklarativnom funkcijom, anonimnu funkciju izrađuje dinamički. Svaki put kad je pozvana, funkcija se dinamički preoblikuje. Ako se funkcija koristi u petlji, ovo znači da se oblikuje sa svakim ponavljanjem petlje, dok bi deklarativna/statička funkcija bila definirana samo jedanput. S obzirom na to mogli biste pomisliti da anonimne funkcije nisu pretjerano korisne. Međutim, dinamička funkcija je odličan način za definiranje funkcionalnosti potrebne za zadovoljavanje potrebe koja se utvrđuje samo u vrijeme izvršavanja.

Evo sintakse anonimne funkcije uz upotrebu konstruktora:

```
var variable = new Function ("parametar1", "parametar2", ... , "parametarn", "tijelo funkcije");
```

Prvi parametri su argumenti za funkciju kako biste ih zadali u deklarativnoj funkciji. Zadnji parametar je tijelo funkcije. Cijela funkcija dodjeljuje se varijabli:

```
var func = new Function("x", "y", "return x * y")
```

To je ekvivalentno sljedećem kodu koji koristi deklarativnu/statičku funkciju:

```
function func (x, y) {
    return x * y;
}
```

Primjer 5-2 dovodi do krajnosti dinamički karakter anonimne funkcije. Korisnik putem dijaloškog okvira unosi sadržaj funkcije i vrijednost njena dva parametra. Cijela se funkcija poziva a rezultat se ispisuje na stranici.

*Primjer 5-2. Dinamička/anonimna funkcija.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Build a Function</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//</pre></div><div data-bbox="679 931 883 949" data-label="Page-Footer"><p>Poglavlje 5: Funkcije | 107</p></div>
```

```

function buildFunction() {

    // Traži funkciju i argumente
    var func= prompt("Enter function body:");
    var x = prompt("Enter value of x:");
    var y = prompt("Enter value of y:");

    // Poziva anonimnu funkciju
    var op = new Function("x", "y", func);
    var theAnswer = op(x, y);

    // Ispisuje rezultat
    alert("Function is: " + func);
    alert( "x is: " + x +
          " y is: " + y);
    alert("The answer is: " + theAnswer);
}

//]]>
</script>
</head>
<body onload="buildFunction();">
<p>Some content</p>
</body>
</html>

```

Budući da je JavaScript slabo tipiziran jezik, funkcija može raditi s bročanim vrijednostima:

```

Function is: return x * y
x is: 33 y is: 11
The answer is: 363

```

Može raditi i s nizovima znakova:

```

Function is: return x + y
x is: This is y is: the string
The answer is: This is the string

```

Jedini zahtjev je da operacija treba imati smisla za odabrani tip podataka. Čak i tada neće doći do JavaScript pogreške jer preglednik ne vidi pogrešku – ona se događa za vrijeme izvršavanja. Rezultat će biti nešto poput ovoga:

```

Function is: return x * y
x is: this is y is: the answers
The answer is: NaN

```

Možete dobiti i neočekivane rezultate. Ako upotrijebite znak plus (+) s dva broja, dobit ćete sljedeći rezultat (umjesto bročanog) jer JavaScript interpretira argumente unutar konteksta kao nizove znakova te ih ulančava:

```

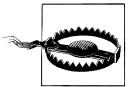
Function is: return x + y;
x is: 2 y is: 3
The answer is: 23

```

Da biste osigurali dobivanje očekivanih rezultata, možete primijeniti eksplicitnu konverziju:

```
Function is: return parseInt(x) + parseInt(y);  
x is: 2 y is: 3  
The answer is: 5
```

Nepotrebno je reći da biste anonimne funkcije trebali koristiti s oprezom. Ne preporučujem da posjetiteljima stranica dopustite da definiraju funkcije koje će se koristiti unutar stranica. No, dinamičke funkcije mogu biti zanimljiv način za rukovanje korisničkim unosima pod uvjetom da iz tog unosa uklonite sve što može izazvati probleme: ugrađene veze, petljanje s kolačićima, pozivanje poslužiteljskih funkcionalnosti, izradu novih funkcija i tako dalje.



Ako naiđete na probleme s primjerom 5-2 u Internet Exploreru 8, možda ćete trebati zadati sigurnosne parametre koji će omogućiti unos podataka u dijaloški okvir.

Jedan hibridni pristup izradi funkcija kombinira statičke mogućnosti deklarativne funkcije s dijelom anonimnosti anonimnih funkcija: literal funkcije opisan u sljedećem odjeljku.

## Literali funkcije

Prije opisa sljedećeg – i potencijalno zbunjujućeg – tipa funkcije, mogao bi biti koristan mali podsjetnik na objekte i doslovne vrijednosti. Kao što sam pokazala u prijašnjim poglavljima, JavaScript objekti mogu biti u obliku doslovne vrijednosti. Umjesto korištenja konstruktora i objekta, možete upotrijebiti reprezentaciju. Niz znakova možete zadati korištenjem konstruktora `String` te mu pristupiti metodama objekta `String`:

```
var str = new String("Learning Java");  
document.writeln(str.replace(/Java/, "JavaScript"));
```

Možete upotrijebiti varijablu temeljenu na osnovnom tipu niza znakova a ipak pristupiti metodama objekta `String` jer JavaScript mehanizam implicitno umotava doslovnu vrijednost u objekt kad se pozove metoda `String`:

```
var str2 = "Learning Java";  
document.writeln(str2.replace(/Java/, "JavaScript"));
```

Ustvari, ne trebate čak niti varijablu:

```
document.writeln("Learning Java".replace(/Java/, "JavaScript"));
```

Ono što vrijedi za nizove znakova, vrijedi i za funkcije, što znači da ne trebate upotrijebiti konstruktor funkcije da biste ju izradili i dodijelili varijabli jer ona doslovno postaje *literal funkcije*:

```
var func = function (parametri) {  
  iskazi;  
}
```

Literali funkcije poznati su i kao *izrazi funkcije* jer je funkcija napisana kao dio izraza umjesto kao zaseban tip iskaza. Oni su nalik na anonimne funkcije po tome što nemaju specifično ime funkcije. No, za razliku od anonimnih funkcija, literali funkcije parsiraju se samo jednom. Ustvari, osim činjenice da se funkcija dodjeljuje varijabli, literali funkcija nalikuju deklarativnim funkcijama:

```
var func = function (x, y) {
    return x * y;
}
alert(func(3,3));
```

Jedinstvenost literala funkcija očita je kad razvijete koncepciju za obavljanje nekog posla, kao što je prosljeđivanje funkcije kao parametra. Primjer 5-3 ilustrira ovo zanimljivo svojstvo. U primjeru je definirana funkcija `funcObject` s tri parametra, od kojih je treći funkcija koja obrađuje prva dva parametra.

*Primjer 5-3. Prosljeđivanje funkcije funkciji.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Function Literal</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//

// Pozivanje trećeg argumenta kao funkcije
function funcObject(x,y,z) {
    alert(z(x,y));
}

function testFunction() {

// Treći parametar je funkcija
funcObject(3,4,function(x,y) { return x * y})
}

//]]&gt;
&lt;/script&gt;
&lt;/head&gt;
&lt;body onload="testFunction();"&gt;
&lt;p&gt;some content&lt;/p&gt;
&lt;/body&gt;
&lt;/html&gt;</pre></div><div data-bbox="115 773 880 813" data-label="Text"><p>Rezultat ove aplikacije je dijaloški okvir s ispisanom vrijednošću "12". Druga inačica poziva funkcije mogla bi biti:</p></div><div data-bbox="154 817 537 835" data-label="Text"><pre>funcObject(4,2,function(x,y) { return x - y})</pre></div><div data-bbox="115 840 880 879" data-label="Text"><p>Rezultat ovog poziva funkcije je oduzimanje parametra <i>y</i> od parametra <i>x</i>, s vrijednošću 2 ispisanom u dijaloškom okviru.</p></div><div data-bbox="115 931 300 949" data-label="Page-Footer"><hr/><p>110 | Naučite JavaScript</p></div>
```

Drugi oblik literala funkcije nije anoniman jer je zadano ime funkcije:

```
var func = function multiply(x,y) {
    return x * y;
}
```

Međutim, *imenu se može pristupiti samo iz sâme funkcije*. To baš nije praktično, osim ako ne implementirate rekurzivnu funkciju.

## Funkcije i rekurzija

Funkcija koja poziva samu sebe naziva se *rekurzivnom funkcijom*. Rekurzivnu funkciju obično koristite kad se proces mora izvršiti više puta, a svako novo ponavljanje procesa koristi rezultat prethodnog procesa. Korištenje rekurzije nije uobičajeno u JavaScriptu, ali može biti korisno kad radite s podacima koji imaju stablastu strukturu, kao što je Document Object Model (DOM). No, to može zahtijevati mnogo memorije i resursa te može biti komplicirano za implementiranje i održavanje. Prema tome, rekurziju koristite štedljivo i svakako sve temeljito testirajte.

U prethodnom dijelu pisala sam o imenovanim literalima funkcije, kad je za funkciju zadano ime, ali mu samo sâma funkcija može pristupiti. To je idealna situacija za rekurziju.

Primjer 5-4 koristi rekurzivnu funkciju za prolaženje kroz brojčano polje, dodavanje brojeva u polje te dodavanje brojeva nizu znakova.

*Primjer 5-4. Rekurzivna JavaScript funkcija.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Recursion</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//
function runRecursion() {

    var addNumbers = function sumNumbers(numArray,indexVal,resultArray) {

        // Test rekurzije
        if (indexVal == numArray.length)
            return resultArray;

        // Zbrajanje brojeva
        resultArray[0] += Number(numArray[indexVal]);

        // Nastavljanje nizova
        if (resultArray[1].length &gt; 0) {
            resultArray[1] += " and ";
        }
        resultArray[1] += numArray[indexVal].toString();</pre></div><div data-bbox="679 931 883 949" data-label="Page-Footer"><hr/><p>Poglavlje 5: Funkcije | 111</p></div>
```

```

    // Povećavanje indeksa
    indexVal++;

    // Ponovno pozivanje funkcije i povrat rezultata
    return sumNumbers(numArray, indexVal, resultArray);
}

// Izrada brojčanog polja i rezultirajuće polje
var numArray = ['1', '35.4', '-14', '44', '0.5'];
var resultArray = new Array(0, ''); // Potrebno za početni slučaj

// Poziv funkcije
var result = addNumbers(numArray, 0, resultArray);

// Rezultat
document.writeln(result[0] + "<br />");
document.writeln(result[1]);
}
//]]>
</script>
</head>
<body onload="runRecursion();">
<p>some content</p>
</body>
</html>

```

Rezultat izvođenja primjera 5-4 je sljedeći prikaz na stranici:

```

66.9
1 and 35.4 and -14 and 44 and 0.5

```

U ovoj JavaScript aplikaciji literal funkcije `sumNumbers` izvodi test da bi se vidjelo je li vrijednost indeksa proslijeđenog kao parametar jednaka dužini polja brojeva, koja je također proslijeđena kao parametar. Ako nije, funkcija dodaje sljedeću brojčanu vrijednost članicu ukupnom zbroju te broj pripaja nizu znakova, od kojih su oboje članovi polja s dva elementa proslijeđenog kao parametar.

Kada je vrijednost indeksa jednaka dužini polja brojeva, funkcija vraća `resultArray`. Budući da je funkcija bila pozvana rekurzivno, rezultat se prosljeđuje kao parametar nakon svakog ponavljanja funkcije `sumNumbers` sve do zadnjeg ponavljanja kad se vrijednost vraća aplikaciji koja je u početku pozvala funkciju putem varijable kojoj je bila dodijeljena.

Naravno, u ovom biste primjeru mogli upotrijebiti petlju `while` za postizanje istih rezultata. No, kao što sam spomenula ranije, kad radite s podacima sa stablastom strukturom kao što je DOM, rekurzija je izuzetno korisna kao i literal funkcije korišten za implementiranje ovog procesa. Međutim, nije svaka upotreba literala funkcije u svim preglednicima bez mogućih negativnih nuspojava. Riskantno područje vezano uz ugniježdene funkcije su moguća curenja memorije izazvana konceptom zvanim *zatvaranje*.

## Ugniježdene funkcije, zatvaranje funkcija i curenja memorije

Još jedan zanimljiv aspekt literala funkcije u JavaScriptu je njihova upotreba kao ugniježdenih funkcija. Proučite sljedeći kôd:

```
function outer (args) {
  function inner (args) {
    inner statements;
  }
}
```

Kod ugniježdenih funkcija unutarnja funkcija djeluje unutar dosega vanjske funkcije te ima pristup njenim varijablama i argumentima. Međutim, vanjska funkcija nema pristup varijablama unutarnje funkcije, niti aplikacija koja poziva ima pristup unutarnjoj funkciji. (Točnije, nema ukoliko nije izrađena kao literal funkcije i vraćena pozivajućoj aplikaciji, što tada dodatno komplicira situaciju.)

Primjer 5-5 prikazuje izradu ugniježđenog literala unutarnje funkcije, koji se zatim vraća pozivajućoj aplikaciji (u primjeru je podebljana). Unutarnja funkcija ima pristup parametru i varijabli vanjske funkcije. Ima i vlastiti parametar kojem vanjska funkcija nema pristup.

Kad se unutarnja funkcija proslijedi aplikaciji putem vanjske funkcije i pozove izravno, ona niz znakova proslijeđen kao parametar u izvornom pozivu vanjske funkcije pripaja nizu znakova koji je proslijeđen unutarnjoj funkciji kao parametar te vraća rezultat. Mijenjanje parametra unutarnje funkcije mijenja niz znakova, kao što to čini i pozivanje vanjske funkcije s drugačijim parametrom te dobivanje druge instance unutarnje funkcije.

*Primjer 5-5. Ugniježdene funkcije i zatvaranje.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Nested</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//

// Vanjska funkcija
function outerFunc(base) {

  var punc = "!";

  // Vraća unutarnju funkciju
  return function (ext) {
    return base + ext + punc;
  }
}

]</pre></div><div data-bbox="679 930 883 949" data-label="Page-Footer"><p>Poglavlje 5: Funkcije | 113</p></div>
```

```

function processNested() {
    // Omogućava pristup unutarnjoj funkciji
    var baseString = outerFunc("Hello ");

    // Unutarnja funkcija i dalje ima pristup argumentu vanjske funkcije
    var newString = baseString("World!");
    alert(newString);

    // I dalje ima pristup
    var notherString = baseString("Reader!");
    alert(notherString);

    // Stvara još jednu instancu unutarnje funkcije
    var anotherBase = outerFunc("Hiya, Hey ");

    // Još jedan lokalni niz
    var lastString = anotherBase("you!");
    alert(lastString);
}
//]]>
</script>
</head>
<body onload="processNested();">
<p>Some content</p>
</body>
</html>

```

Rezultat ove aplikacije su tri dijaloška okvira koja se otvaraju jedan za drugim i poručuju „Hello World!“, „Hello Reader!“ i „Hiya, Hey you!“

Kako ovo radi? Nije li to kršenje pravila dosega koje kaže da, kada funkcija završi, sva se memorija potrošena za njene lokalne varijable oslobađa automatskim sakupljanjem otpada?

Ne sasvim.

Svaki put kad se definira novi doseg u JavaScript aplikaciji, stvara se pridruženi *mje-hurić dosega*, da ga tako nazovemo, kako bi ga obuhvatio. To vrijedi za funkcije, koje djeluju u vlastitom dosegu.

U pravilu, kad funkcija završi, doseg se oslobađa jer više nije potreban. Međutim, u slučaju unutarnje funkcije koja je vraćena vanjskoj aplikaciji i dodijeljena vanjskoj varijabli, doseg unutarnje funkcije pripaja se vanjskoj funkciji, koja je sa svoje strane pridružena aplikaciji koja poziva – upravo dovoljno za održavanje integriteta unutarnje funkcije te argumenta i varijable vanjske funkcije. Vraćanje literala funkcije izrađenog kao unutarnji objekt unutar druge funkcije te njegovo dodjeljivanje varijabli u aplikaciji koja poziva u JavaScriptu je poznato kao *zatvaranje* (engl. *closure*). Ova koncepcija ulančavanja dosega osigurava da će podaci potrebni za rad aplikacije biti na svom mjestu.

To je važna koncepcija u JavaScript bibliotekama pa ćemo se zatvaranju vratiti kasnije u knjizi. No, postoji i problem vezan uz zatvaranje – možete to učiniti slučajno. U primjeru 5-5, ako se izradi nova referenca na unutarnju funkciju za svaki izrađeni niz znakova, umjesto da se ponovno upotrijebi varijabla koja pruža referencu na unutarnju funkciju, tijekom vremena nastat će mnoge instance tog objekta.

Do slučajnog zatvaranja može doći kad se izradi kružna referenca, kao što je sljedeća sa Mozillinih stranica s dokumentacijom:

```
function leakMemory() {  
    var e1 = document.getElementById('e1');  
    var newObj = { 'e1': e1 };  
    e1.newObj = newObj;  
}
```

DOM ćemo detaljnije proučiti u poglavlju 11, a metodu `getElementById` uvelike ćemo koristiti u drugoj polovini knjige. Ali, u ovom primjeru koda DOM-u se pristupa radi uzimanja elementa identificiranog s `e1`. Element se zatim koristi za izradu nove reference na objekt (`newObj`) u formatu koji će također opisati kasnije u knjizi. Novi literal funkcije pravi neimenovani objekt koji dobiveni DOM objekt dodjeljuje svojstvu `e1`.

Ovdje dolazi do obrata: `newObj` dodjeljujemo upravo dodanom svojstvu na izvornom elementu dobivenom od DOM-a (savršeno prihvatljiv postupak u JavaScriptu), što doslovce znači da smo objekt dodijelili kao njegovo vlastito svojstvo. Ovo nije nešto što bih željela ohrabrivati, ali većina preglednika uspijeva prekinuti zatvaranje i osloboditi memoriju, osim starijih inačica Internet Explorera.

Starije (ali još uvijek u upotrebi) inačice IE-a (6.x i 7.x) imaju vlastito upravljanje memorijom za DOM objekte pored upravljanja memorijom za JavaScript objekte. U slučaju nenamjernih zatvaranja koja su izazvale kružne reference poput ovih iz primjera te prijelaza između JavaScripta i DOM objekata, memorija se alocira i uopće se ne oslobađa, čak ni kada se stranica zatvori. Ustvari, memorija se oslobađa samo *kad se zatvori preglednik*.

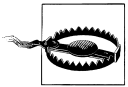
Curenje memorije do kojeg dolazi obično je malo, osim ako sve to ne stavite u petlju, a u tom se slučaju gubitak memorije brzo može nagomilati. Zbog problema s curenjem memorije i rasprostranjenosti starijih inačica IE-a trebali biste biti oprezni pri korištenju zatvaranja, kao što sam pokazala u ovom odjeljku. Srećom, problem curenja memorije je riješen u Internet Exploreru 8.



Zatvaranje je odlično objašnjeno u dokumentima Jima Leya, dostupnim na adresi [http://jibbering.com/faq/faq\\_notes/closures.html](http://jibbering.com/faq/faq_notes/closures.html).

## Funkcije povratnog poziva

U dijelu „JavaScript polja“ u poglavlju 4 napisala sam da neke metode ovise o funkcijama koje se pozivaju automatski na temelju nekog događaja. Metode objekta Array su `filter`, `forEach`, `every`, `map` i `some`, a funkcije koje se koriste su literali funkcije, iako se prilikom ovakve upotrebe obično nazivaju *funkcijama povratnog poziva* (engl. *callback functions*).



Funkcije povratnog poziva objekta Array nisu dio standarda ECMAScript. Iako primjer u ovom dijelu radi u Safariju, Operi i Firefoxu, ne radi u Internet Exploreru, uključujući i najnoviji IE8.

Nadalje, što se tiče metoda objekta Array, metoda `filter` osigurava da elementi ne budu dodani nijednom polju osim ako ne udovolje određenim kriterijima. Umjesto potrebe za testiranjem vrijednosti i zatim pojedinačnim dodavanjem polju ako je element prošao filter koji ste primijenili, jednostavno možete sve staviti u polje i dopustiti metodi `filter` da obavi posao umjesto vas.

Metoda `forEach` uzima funkciju koja obrađuje svaki element. Za razliku od metode `filter`, ova funkcija ne utječe na polje. Metoda `map` izvodi funkciju povratnog poziva na svim elementima polja i od rezultata pravi novo polje. Slična je metodi `forEach` po tome što se povratni poziv izvodi za svaki element polja. Međutim, `forEach` prvenstveno koristite za obradu funkcije za svaki element polja, a `map` koristite za izradu novog polja s rezultatima izvođenja funkcije sa svakim elementom polja.

Metoda `every` izvodi funkciju povratnog poziva za svaki element u polju sve dok jedan element ne vrati vrijednost `false`. Konačno, metoda `some` suprotna je od `every` jer izvodi funkciju povratnog poziva za sve elemente sve dok jedan ne vrati `true`.

Sve funkcije povratnog poziva uzimaju tri parametra: `element`, `index` i `array`. Neke vraćaju vrijednost a neke ne. Nijedna ne utječe na izvorno polje.

Primjer 5-6 pokazuje kako koristiti funkciju povratnog poziva na objektu Array. U ovom primjeru izvorno polje sadrži elemente koji su i sami polje s vrijednostima boja, ali nekoliko „boja“ ima vrijednost izvan dopuštenog raspona od 0 do 255. Nakon što se polje oblikuje, pridružuje se jedna funkcija, `checkColor`, koja provjerava svaki element polja za odgovarajući raspon boja te filtrira one koji se ne uklapaju u raspon.

Ovo vodi do drugog problema jer nekoliko polja boja više nemaju tri potrebna elementa za postavke crvene, zelene i plave boje. Zatim se na polje primjenjuje druga funkcija povratnog poziva radi osiguranja da su prisutne tri RGB vrijednosti.

*Primjer 5-6. Korištenje funkcija povratnog poziva s metodom filter objekta Array.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Callbacks</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

```

<script type="text/javascript">
//

// Provjerava funkciju povratnog poziva polja boja
function checkColor(element,index,array) {
    return (element &gt;= 0 &amp;&amp; element &lt; 256);
}

// Provjerava postoje li sve tri RGB komponente boje
function checkCount(element,index,array) {
    return (element.length == 3);
}

function testingCallbacks() {

    // Polje boja
    var colors = new Array();
    colors[0] = [0,262,255];
    colors[1] = [255,255,255];
    colors[2] = [255,0,0];
    colors[3] = [0,255,0];
    colors[4] = [0,0,255];
    colors[5] = [-5,999,255];
    colors[6] = [255,255,1204556];

    // Filter na polju boja
    var testedColors = new Array();
    for (var i in colors) {
        testedColors[i] = colors[i].filter(checkColor);
    }

    // Ispisuje rezultate prvog kruga
    document.writeln("&lt;h3&gt;First check&lt;/h3&gt;");
    for (i in testedColors) {
        document.writeln(testedColors[i] + "&lt;br /&gt;");
    }

    // Izdvaja zapise sa manje od tri vrijednosti
    var newTested = testedColors.filter(checkCount);
    document.writeln("&lt;br /&gt;&lt;h3&gt;Second&lt;/h3&gt;");

    // Ispisuje rezultate
    for (i in newTested) {
        document.writeln(newTested[i] + "&lt;br /&gt;");
    }
}
//]]&gt;
&lt;/script&gt;
&lt;/head&gt;
&lt;body onload="testingCallbacks();"&gt;
&lt;p&gt;Some content&lt;/p&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>
</div>
<div data-bbox="117 862 883 900" data-label="Text">
<p>Na kraju, samo četiri boje prolaze obje provjere – od drugog do petog elementa polja, kao što prikazuje ispis rezultata:</p>
</div>
<div data-bbox="679 931 883 949" data-label="Page-Footer">
<hr/>
<p>Poglavlje 5: Funkcije | 117</p>
</div>
```

```
First check
0,255
255,255,255
255,0,0
0,255,0
0,0,255
255
255,255
```

```
Second
255,255,255
255,0,0
0,255,0
0,0,255
```

Iako funkcije povratnog poziva pojednostavljuju aplikacije, one nisu obavezne. Za testiranje članova polja možete koristiti kontrolne petlje. Ako aplikacija treba raditi i u Internet Exploreru, morat ćete koristiti kontrolne petlje.

## Sažeti opis tipova funkcija

Da rezimiramo, postoje tri različita tipa funkcija:

### *Deklarativna funkcija*

Funkcija u vlastitom iskazu koja započinje ključnom riječi `function`. Deklarativne funkcije se parsiraju jedanput, statičke su i zadano im je ime preko kojeg im se pristupa.

### *Anonimna funkcija*

Funkcija izrađena pomoću konstruktora. Parsira se svaki put kad joj se pristupi i nije joj posebno zadano ime.

### *Literal funkcije ili izraz funkcije*

Funkcija izrađena unutar drugog iskaza kao dio izraza. Parsira se jedanput, statička je i može joj ili ne mora biti zadano specifično ime. Ako je imenovana, imenu se može pristupiti jedino iz same funkcije.

Deklarativne funkcije dostupne su u svim inačicama JavaScripta u svim preglednicima. Anonimne funkcije temeljene na konstruktorima funkcija dinamičke su, zahtjevne za memoriju i temelje se na novijim inačicama JavaScripta pa stoga možda nisu dostupne u starijim preglednicima. Literal funkcije noviji su izum, temeljen na JavaScriptu 1.5. Podržavaju ih samo najmoderniji preglednici, među kojima su i oni najpopularniji: Mozilla, Firefox, IE i Safari. Međutim, način na koji svaki od njih radi s literalima funkcije može dovesti do zanimljivih komplikacija u korištenju memorije, kao što ste naučili ranije u ovom poglavlju, u odjeljku „Ugniježdene funkcije, zatvaranje funkcija i curenja memorije“.

Literal funkcije također tvore osnovu za većinu naprednih Ajax biblioteka, kao što ćete vidjeti kad pomnije proučimo Ajax i napredno korištenje JavaScripta u zadnjih nekoliko poglavlja knjige. Osim toga, literalne funkcije koristite s metodama za obradu

događaja objekta koje zahtijevaju funkcije povratnog poziva, poput onih pridruženih objektu `Array`.

Kad već spominjemo objekte, ranije je rečeno da su funkcije u JavaScriptu također objekti, osim što su nizovi iskaza. U zadnjem dijelu ovog poglavlja detaljnije ću obraditi ovaj koncept. No, prvo ukratko proučimo doseg funkcije i njegov utjecaj na globalne i lokalne varijable.

## Doseg funkcije

U poglavlju 2 spomenula sam globalne i lokalne varijable te pojam *dosega* varijable. Doseg je zapravo vezan uz mogućnost pristupa varijabli izvan dosega funkcije.

Korištenje ključne riječi `var` s varijablom u funkciji govori aplikaciji koja obrađuje JavaScript da je varijabla deklarirana s lokalnim dosegom, što znači da je dostupna samo unutar bloka koda funkcije. Izvan funkcije, varijabla nije definirana:

```
function test () {
  var myTest = "test";
  ...
  alert(myTest); // Ispisuje "test"
}
alert(myTest); // undefined
```

Ključnu riječ `var` možete upotrijebiti s varijablom izvan funkcije, ali bez obzira na to koristite li ključnu riječ ili ne, varijabla ima globalni doseg jer nije sadržana unutar funkcije.

Pored toga, zaboravite li navesti ključnu riječ `var` za varijablu unutar funkcije, aplikacija koja obrađuje JavaScript smatra da je to globalna varijabla i s njome postupaju sukladno tome:

```
function test() {
  myTest = "test";
  ...
  alert(myTest); // Ispisuje "test"
}
alert(myTest); // Ispisuje "test"
```

U manjim aplikacijama globalna varijabla ne mora prouzročiti nikakvu štetu. No, kod većih aplikacija druge globalne varijable mogle bi koristiti isto ime i u tom bi slučaju upotreba globalne varijable mogla prepisati važne informacije. Nasuprot tome, i vaše važne informacije mogle bi biti prepisane.

Izvedba je također problem. Kad se varijabla deklarira lokalno i funkcija završi, resursi korišteni za tu varijablu oslobađaju se u procesu poznatom kao *sakupljanje otpada* te se stavljaju na raspolaganje za druge potrebe aplikacije. No, globalne varijable se ne oslobađaju sve dok cijela aplikacija ne završi.

Ukratko, zapamtite da s varijablama u funkcijama trebate koristiti ključnu riječ `var` i izbjegavati globalne varijable. Ako su vam potrebni globalno pristupačni podaci,

mogli biste izraditi prilagođeni objekt za aplikaciju i zatim mu dodijeliti sve potrebne globalne vrijednosti. Ovo barem pruža neku kontrolu nad brojem globalnih varijabli koje se nalaze u aplikaciji. Prilagođene objekte detaljno ću obraditi u poglavlju 13.

Novije i buduće inačice JavaScripta čak će omogućavati definiranje varijable unutar dosega kontrolnog bloka, kao što je petlja `while`. No, umjesto korištenja ključne riječi `var` upotrijebili biste ključnu riječ `let` kao u sljedećem kodu:

```
function myTest() {
  var someVar = 1;
  while(someTest) {
    let someVar = 2; // Druga varijabla
  }
}
```

Međutim, korištenje ključne riječi `let` i zadavanja dosega na razini bloka vrlo je novo i nije široko implementirano pa se na njega još ne bih oslanjala.

## Funkcija kao objekt

Što god možete izraditi pomoću konstruktora ima svojstva i metode iznad i izvan očitih, a funkcije nisu izuzetak.

Čini se da je JavaScript objekt `Function` tijekom vremena prošao kroz najviše promjena. Izvorno, svojstvo `arity` pružalo je broj argumenata. Ono je bilo zamijenjeno pozivanje metode `length` izvan imena funkcije ili pristupanjem svojstvu `length` na polju argumenata. Ono samo bilo je pristupačno putem imena funkcije, ali tada je promijenjeno tako da bude pristupačno kao `arguments` unutar poziva funkcije.

Jedina standardna svojstva i metode dopušteni za objekt `Function`, osim onih koji su standardni za sve objekte, poput `toSource` i `toString`, su `length` za broj argumenata, te dvije metode, `apply` i `call`, koje ću detaljnije opisati u poglavlju 13.

Osim toga, kad se izrađuju prilagođeni objekti, važna za izradu klasa novih objekata je mogućnost funkcije da izradi reference na vlastiti doseg pomoću specijalizirane ključne riječi `this`.

---

## Testirajte znanje: pitanja

1. Faktorijel je broj  $n$ , umnožak svih brojeva od 1 do  $n$ , obično napisan kao  $3!$  ( $1 \times 2 \times 3$  ili  $6$ ). Napišite JavaScript funkciju koja koristi rekurziju za izračunavanje faktorijela broja sa zadanim brojem.

2. Kako funkcija može mijenjati varijable izvan svoga dosega? Napišite funkciju koja uzima polje brojeva od 1 do 5 i zamjenjuje stavke sa znakovnim reprezentacijama brojeva (tj. „one“, „two“ itd.).
3. Napišite funkciju koja kao parametre uzima objekt podataka i funkciju te poziva funkciju koristeći objekt podataka.

## Testirajte znanje: odgovori

1. Jedno rješenje je:

```
function findFactorial(n) {  
    if (n == 0) return 1;  
    return (n * findFactorial(n-1));  
}  
  
var num = findFactorial(4); // Vraća 24
```

2. Ako je objekt, kao što je polje, prosljeđen kao parametar funkcije, promjene u polju se općenito odražavaju izvan funkcije. Jedno od rješenja za kôd funkcije iz pitanja je sljedeće:

```
function makeArray() {  
    var arr = [1,5,3];  
    alert(arr);  
    alterArray(arr);  
    alert(arr);  
}  
  
function alterArray(arrOfNumbers) {  
    for (var i = 0; i < arrOfNumbers.length; i++) {  
        switch(arrOfNumbers[i]) {  
            case 1 : arrOfNumbers[i] = "one"; break;  
            case 2 : arrOfNumbers[i] = "two"; break;  
            case 3 : arrOfNumbers[i] = "three"; break;  
            case 4 : arrOfNumbers[i] = "four"; break;  
            case 5 : arrOfNumbers[i] = "five"; break;  
        }  
    }  
}
```

3. Anonimna funkcija udovoljava ovim zahtjevima:

```
function invokeFunction(dataObject, functionToCall) {  
    functionToCall(dataObject);  
}  
var funcCall = new Function('x', 'alert(x)');  
invokeFunction ('hello', funcCall);
```