POGLAVLJE 27

Sintaksa

Ako ste pisali JS kôd barem neko vreme, vrlo je verovatno da vam je sintaksa prilično poznata. Svakako postoje mnoge posebnosti, ali u opštem slučaju, to je prilično smislena i lako razumljiva sintaksa koja ima mnogo sličnosti s drugim jezicima.

Međutim, ES6 uvodi priličan broj novih sintaksnih oblika na koje se treba navići. U ovom poglavlju ćemo ih navesti pojedinačno da bismo razmotrili šta sve postoji.

U vreme pisanja ove knjige, neke od mogućnosti koje razmatramo već su implementirane u raznim čitačima veba (Firefox, Chrome itd.), dok su druge implementirane samo delimično, a mnoge uopšte nisu implementirane. Vaše lično iskustvo koje ćete imati ako isprobate primere iz ove knjige može biti drugačije od onog što je ovde opisano. U takvim slučajevima, isprobajte ih u verziji s transpajlerima, pošto te alatke pokrivaju naveći deo novih ES6 mogućnosti.

ES6Fiddle (*http://www.es6fiddle.net/*) odlično i lako upotrebljivo „igralište“ za isprobavanje ES6 koda, kao što je veb REPL za transpajler Babel (*http://babeljs.io/repl/*).

Deklaracije čiji je opseg vidljivosti blok

Verovatno znate da je u JavaScriptu osnovna jedinica za opseg vidljivosti promenljivih oduvek bila funkcija. Ako vam je trebalo da formirate opseg vidljivosti čija je veličina određeni blok koda, najuobičajeniji način da to uradite – osim korišćenja standardne funkcije – bio je upotreba *funkcijskog izraza za trenutno izvršavanje* (IIFE). Na primer:

```
var a = 2;

(function IIFE(){
    var a = 3;
    console.log( a );    // 3
})();

console.log( a );        // 2
```

Deklarisanje promenljivih pomoću rezervisane reči let

Međutim, sada možemo zadavati deklaracije promenljivih koje su vezane za određeni blok, što se (nimalo iznenađujuće) zove *opseg vidljivosti veličine bloka*. To znači da nam treba samo par { .. } kako bismo definisali opseg vidljivosti. Umesto rezervisane reči `var`, koja uvek deklariše promenljive čiji je opseg vidljivosti okružujuća funkcija (ili globalni opseg, ako su deklarisane na najvišem nivou), zadajte rezervisanu reč `let`:

```
var a = 2;

{
    let a = 3;
    console.log( a );    // 3
}

console.log( a );        // 2
```

U JS-u dosad nije bila mnogo uobičajena upotreba samostalnog bloka { .. }, ali je to oduvek bila ispravna sintaksa. Taj model će odmah prepoznati programeri iz drugih jezika u kojima se može definisati *opseg vidljivosti veličine bloka*.

Verujem da je to najbolji način da se deklarišu promenljive čiji opseg vidljivosti je blok, unutar vlastitog bloka { .. }. Osim toga, trebalo bi da `let` deklaraciju (ili deklaracije) uvek postavite na početak tog bloka. Ako treba da deklarišete više promenljivih, preporučio bih vam da zadate samo jedan `let`.

Što se stila pisanja tiče, ja čak najradije pišem `let` u istom redu gde je početna zagrada {, kako bih jasnije istakao da je svrha tog bloka samo deklarisanje opsega vidljivosti tih promenljivih.

```
{   let a = 2, b, c;
    // ..
}
```

To će izgledati čudno i verovatno se neće slagati s preporukama iz većeg dela literature o ES6. Ali imam dobre razloge za svoju uvrnutost.

Postoji još jedan eksperimentalan (nestandardizovan) oblik `let` deklaracije nazvan `let` blok, koji izgleda ovako:

```
let (a = 2, b, c) {
    // ..
}
```

Taj oblik je ono što zovem *eksplicitni* opseg vidljivosti veličine bloka, dok je oblik `let ..` deklaracije koji preslikava oblik `var` deklaracije više *implicitan*, pošto nije najjasnije vidljivo kojem paru { .. } pripada. Programeri uglavnom smatraju da je upotreba *eksplicitnih* mehanizama ipak preporučljivija od *implicitnih* mehanizama, a ja tvrdim da je ovo jedan od takvih slučajeva.

Ako uporedite prethodna dva oblika, oni su vrlo slični, a po mom mišljenju, oba po stilu pripadaju *eksplicitnom* zadavanju opsega vidljivosti veličine bloka. Nažalost, oblik

`let (..) { .. }`, *najeksplicitnija* opcija, nije prihvaćen u verziji ES6. To će se možda promeniti u nekoj verziji posle ES6, ali mislim da nam je prva opcija zasad najbolja.

Da biste bolje shvatili *implicitnu* prirodu `let ..` deklaracija, razmotrite sledeće slučajeve upotrebe:

```
let a = 2;

if (a > 1) {
    let b = a * 3;
    console.log( b );       // 6

    for (let i = a; i <= b; i++) {
        let j = i + 10;
        console.log( j );
    }
    // 12 13 14 15 16

    let c = a + b;
    console.log( c );       // 8
}
```

Brza provera, bez gledanja u primer: koja promenljiva (ili promenljive) postoji samo unutar iskaza `if`, a koja promenljiva (ili promenljive) postoji samo unutar petlje `for`?

Odgovori: iskaz `if` sadrži promenljive `b` i `c` čiji je opseg vidljivosti blok iskaza `if`, a petlja `for` sadrži promenljive `i` i `j` čiji je opseg vidljivosti blok petlje.

Da li ste morali da razmišljate izvesno vreme? Iznenađuje li vas to što promenljiva `i` nije dodata u opseg vidljivosti bloka svog okružujećeg iskaza `if`? Razlog te mentalne pauze i pitanja – ja to zovem „mentalna taksa“ – potiče od činjenice da taj `let` mehanizam ne samo što je za nas novina, nego je i *implicitan*.

Činjenica da se deklaracija `let c = ..` nalazi tako duboko unutar opsega vidljivosti uvodi i određenu opasnost. Za razliku od promenljivih koje deklarišete na tradicionalan način pomoću `var`, koje važe u celom okružujućem opsegu vidljivosti funkcije u kojem su deklarisane, `let` deklaracije se dodaju u opseg vidljivosti svog bloka, ali se ne inicijalizuju sve do mesta gde se pojavljuju u bloku.

Pristupanje promenljivoj koja je deklarisana pomoću `let` pre njene `let ..` deklaracije/inicijalizacije prouzrokuje grešku, dok je za `var` deklaracije taj redosled nevažan (osim u pogledu stila).

Razmotrite sledeće:

```
{
    console.log( a );    // undefined
    console.log( b );    // ReferenceError!

    var a;
    let b;
}
```

Ta greška `ReferenceError` zbog preranog pristupanja promenljivoj koja je deklarisana pomoću `let` tehnički se zove *TDZ (Temporal Dead Zone – privremena mrtva zona)* greška – pristupate promenljivoj koja je deklarisana ali još nije inicijalizovana. To nije jedini slučaj kada viđamo TDZ greške – one se u ES6 pojavljuju na više mesta. Osim toga, imajte u vidu da „inicijalizovana" ne znači eksplicitno dodeljivanje vrednosti u kodu, pošto je `let b;` potpuno ispravna sintaksa. Budući da se za promenljivu kojoj u trenutku deklarisanja nije dodeljena nikakva vrednost podrazumeva da joj je dodeljena vrednost `undefined`, iskaz `let b;` isto je što i `let b = unde fined;`. Bez obzira na to da li je inicijalizujete eksplicitno ili implicitno, promenljivoj `b` ne možete pristupati dok se ne izvrši iskaz `let b`.

Poslednja zamka: operator `typeof` se sa TDZ promenljivama ponaša drugačije nego s nedeklarisanim (ili deklarisanim!) promenljivama. Na primer:

```
{
    // `a` nije deklarisana
    if (typeof a === "undefined") {
        console.log( "cool" );
    }

    // `b` je deklarisana, ali u svojoj TDZ
    if (typeof b === "undefined") { // ReferenceError!
        // ..
    }

    // ..

    let b;
}
```

Pošto promenljiva `a` nije deklarisana, upotreba operatora `typeof` jedini je bezbedan način da proverimo postoji li ona ili ne. Ali, izraz `typeof b` prouzrokuje TDZ grešku zato što duboko niže u kodu slučajno imamo deklaraciju `let b`. Uh.

Sada bi trebalo da bude jasnije zbog čega insistiram da se sve `let` deklaracije nalaze na početku svojih opsega vidljivosti. Tako se potpuno izbegavaju nenamerne greške zbog preranog pristupanja promenljivoj. Osim toga, tako postaje *eksplicitnije* – kada pogledate početak bloka, svakog bloka – koje promenljive on sadrži.

Vaši blokovi (iskazi `if`, petlje `while` itd.) ne moraju da dele svoje izvorno ponašanje s ponašanjem u vezi sa opsegom vidljivosti.

Ta vaša eksplicitnost, koju ćete samo morati da disciplinovano održavate, na duži rok će vas poštedeti mnogih glavobolja.

Više informacija o deklarisanju promenljivih pomoću `let` i opsega vidljivosti veličine bloka naći ćete u drugom delu ove knjige – *Opseg vidljivosti i ograde*.

Deklaracija let i petlja for

Jedini izuzetak koji bih napravio u vezi s *eksplicitnim* oblikom deklaracije `let` u bloku jeste `let` deklaracija koja se nalazi u zaglavlju petlje `for`. Razlog će vam možda izgledati nejasan, ali lično verujem da je to jedna od najvažnijih ES6 odlika.

Razmotrite sledeće:

```
var funcs = [];

for (let i = 0; i < 5; i++) {
    funcs.push( function(){
        console.log( i );
    } );
}

funcs[3]();     // 3
```

Izraz `let i` u zaglavlju petlje `for` deklariše promenljivu `i` ne samo za petlju `for`, nego ponovo deklariše novu promenljivu `i` u svakoj iteraciji petlje. To znači da ograde koje se formiraju unutar jedne iteracije petlje ograđuju te promenljive po iteraciji, kao što biste i očekivali.

Ako isprobate isti primer, ali s deklaracijom `var i` u zaglavlju petlje `for`, rezultat će biti `5` umesto `3`, zato što bi u spoljašnjem opsegu koji funkcija ograđuje postojala samo jedna promenljiva `i`, umesto nove promenljive `i` u svakoj iteraciji koju bi funkcija ogradila.

Isto možete postići i u neznatno opširnijem obliku:

```
var funcs = [];

for (var i = 0; i < 5; i++) {
    let j = i;
    funcs.push( function(){
        console.log( j );
    } );
}

funcs[3]();     // 3
```

U ovom slučaju, izričito deklarišemo novu promenljivu `j` u svakoj deklaraciji, a nakon toga ograda deluje na isti način. Draže mi je prvo rešenje; zbog dodatne mogućnosti koju ono pruža radije se opredeljujem za oblik `for (let .. ) ...`. Može se opravdano zameriti da je to malo *implicitniji* oblik, ali je za moj ukus ipak dovoljno *eksplicitan* i koristan.

Deklaracija pomoću `let` ponaša se na isti način i u petljama `for..in` *i* `for..of` (videti odeljak „Petlja for..of“ na strani 693).

Deklarisanje konstanti

Postoji još jedna deklaracija čiji opseg važenja je blok: to je deklaracija pomoću rezervisane reči `const`, čime se formiraju *konstante*.

Šta je tačno konstanta? To je promenljiva čija se vrednost nakon inicijalizovanja može samo čitati. Razmotrite sledeće:

```
{
    const a = 2;
    console.log( a );    // 2

    a = 3;               // greška TypeError!
}
```

Nakon inicijalizovanja vrednosti promenljive, u trenutku deklarisanja, više nije dozvoljeno menjati tu vrednost. Deklaracija `const` mora sadržati i eksplicitnu operaciju inicijalizovanja vrednosti promenljive. Ako želite da vrednost *konstante* bude `undefined`, morate deklarisati `const a = undefined`.

Konstante ni po čemu ne ograničavaju samu vrednost, nego samo dodeljivanje vrednosti konstanti. Drugim rečima, sama vrednost nije zbog deklaracije `const` postala ni „zamrznuta“, ni nepromenljiva, nego to važi samo za operaciju dodeljivanja vrednosti. Ako je to neka složena vrednost, kao što je objekat ili niz, sadržaj te vrednosti se i dalje može menjati:

```
{
    const a = [1,2,3];
    a.push( 4 );
    console.log( a );    // [1,2,3,4]

    a = 42;              // greška TypeError!
}
```

Promenljiva `a` ne sadrži zaista konstantan niz, nego konstantnu referencu na niz. Sam niz se slobodno može menjati.

Dodeljivanje objekta ili niza kao vrednost konstante znači da sakupljač smeća neće moći da pokupi tu vrednost sve dok leksički opseg vidljivosti te konstante ne nestane, zato što se nikad ne može izbrisati referenca na tu vrednost. To može biti poželjno, ali vodite računa ako vam to nije bila namera!

U suštini, deklaracija pomoću `const` sada samo nameće ono što smo godinama ugrađivali u kôd kao stilski oblik, gde smo ime promenljive deklarisali velikim slovima i dodeljivali joj neku literalnu vrednost za koju smo vodili računa da je nakon toga nikad ne menjamo. Nema ograničenja za promenljive deklarisane pomoću `var`, ali ima za promenljive deklarisane pomoću `const`, što može da vam pomogne da otkrijete nenamerne izmene vrednosti.

Rezervisana reč `const` *može* se upotrebiti za deklarisanje promenljivih u petljama `for`, `for..in` i `for..of` (videti odeljak „Petlja for..of“ na strani 693). Međutim, svaki pokušaj dodeljivanja nove vrednosti, kao što je u tipičnoj odredbi `i++` petlje `for`, prouzrokovaće grešku.

Da li koristiti const ili ne

Postoje mišljenja da bi u nekim slučajevima JS mašina mogla da optimizuje `const` deklaraciju bolje od `let` ili `var` deklaracije. Teorijski, tada mašina jezika lakše prepoznaje da se vrednost/tip promenljive neće nikad menjati, pa zato može da eliminiše moguć režijski deo posla.

Bez obzira na to da li je `const` deklaracija zaista korisna ili su to samo naše fantazije i intuicije, važnija odluka koju treba doneti je da li namerno želite ponašanje konstante ili ne. Ne zaboravite: jedna od najvažnijih uloga vašeg izvornog koda jeste da jasno pokazuje šta je tačno vaša namera, i to ne samo vama u ovom trenutku, nego i vama ubuduće, kao i drugima koji sarađuju s vama na tom kodu.

Neki programeri radije prvo deklarišu svaku promenljivu kao `const` a zatim ublažavaju deklaraciju u `let` ako se u kodu pojavi potreba da se njena vrednost menja. To je zanimljiva ideja, ali nije jasno da li to zaista poboljšava čitljivost ili razumljivost koda.

To nije nekakva *zaštita*, kao što mnogi veruju, zato što svaki programer koji naknadno poželi da promeni vrednost promenljive deklarisane kao `const` može jednostavno u njenoj deklaraciji `const` izmeniti u `let`. U najboljem slučaju, to štiti od slučajnih izmena. Ali ponovo, izvan naših intuicija i gledišta, izgleda da ne postoji nikakva objektivna i jasna mera šta je tačno „slučajno“ i kako to predupredi. Slična oprečna mišljenja postoje i u vezi s konverzijom tipova.

Moj savet: da biste izbegli potencijalno zbunjujući kôd, koristite `const` isključivo za deklarisanje promenljivih za koje namerno i očigledno želite da istaknete da se njihove vrednosti neće menjati. Drugim rečima, ponašanje koda nemojte *zasnivati* na `const`, nego tu vrstu deklaracije koristite kao alatku da biste signalizirali nameru, kada se ta namera može jasno pokazati.

Funkcije čiji je opseg vidljivosti blok

Počev od ES6, opseg vidljivosti funkcija čije se deklaracije nalaze unutar blokova ograničen je na blok u kojem su deklarisane. Pre ES6, to nije bilo izričito zadato u specifikaciji, ali su mnoge implementacije tako radile, pa je sada specifikacija usklađena sa stvarnošću.

Razmotrite sledeće:

```
{
    foo();                  // radi!

    function foo() {
        // ..
    }
}

foo();                      // ReferenceError
```

Funkcija `foo()` je deklarisana unutar bloka `{ .. }` i za ES6 njen opseg vidljivosti je taj blok. Dakle, ona nije vidljiva izvan tog bloka. Ali obratite pažnju i na to da je deklaracija funkcije „podignuta“ na početak bloka, za razliku od `let` deklaracija, koje uvode već ranije pomenutu zamku TDZ greške.

Deklarisanje funkcija čiji je opseg vidljivosti blok može biti problem ako ste navikli da pišete kôd kao što je sledeći i očekujete ponašanje kao ranije, kad opseg vidljivosti nije bio blok:

```
if (something) {
    function foo() {
        console.log( "1" );
    }
}
else {
    function foo() {
        console.log( "2" );
    }
}

foo();    // ??
```

U okruženjima starijim od ES6, funkcija `foo()` bi uvek ispisala `"2"` bez obzira na vrednost `something`, zato što su se obe deklaracije funkcije izvlačile iz svojih blokova i uvek bi važila samo poslednja deklaracija.

U ES6, poslednji red izaziva grešku `ReferenceError`.

Operator za razdvajanje/grupisanje ostatka vrednosti

ES6 uvodi nov operator `...` koji se obično naziva operator za *razdvajanje* ili operator za *grupisanje ostatka* vrednosti (engl. *spread* ili *rest operator*), zavisno od toga gde/kako je upotrebljen. Pogledajmo kako to izgleda:

```
function foo(x,y,z) {
    console.log( x, y, z );
}

foo( ... [1,2,3] );            // 1 2 3
```

Kada operator `...` zadate ispred nekog niza (zapravo, ispred svakog *iterabilnog objekta*, što ćemo razmotriti u poglavlju 28), on se ponaša tako da „razdvaja“ niz na njegove pojedinačne elemente.

Na slučajeve upotrebe u obliku kao što je prikazan u primeru, nailazićete kada treba razdvojiti zadati niz na skup argumenata u pozivu funkcije. Operator `...` se u tom slučaju ponaša kao jednostavnija sintaksna zamena metode `apply(..)`, koju bismo pre ES6 najčešće upotrebili u sledećem obliku:

```
foo.apply( null, [1,2,3] ); // 1 2 3
```

Ali, operator `...` može se iskoristiti za razdvajanje/izdvajanje vrednosti i u drugim kontekstima – recimo, unutar deklaracije niza:

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

console.log( b );    // [1,2,3,4,5]
```