POGLAVLJE 10

Sada this postaje jasnije!

U poglavlju 9 raščistili smo s raznim pogrešnim mišljenjima o identifikatoru `this` i saznali da je `this` veza koja se uspostavlja pri svakom pozivanju funkcije i zavisi isključivo od mesta pozivanja funkcije (načina na koji je ona pozvana).

Mesto pozivanja

Da bismo razumeli kako se `this` povezuje (tj. šta u datom trenutku predstavlja), moramo shvatiti *mesto pozivanja* (engl. *call-site*): to je mesto u kodu gde se funkcija poziva (to nije mesto gde je ona deklarisana). Moramo ispitati mesto pozivanja funkcije da bismo odgovorili na pitanje: šta *konkretno* `this` predstavlja?

Pronalaženje mesta pozivanja uglavnom se svodi na „pronaći gde se funkcija poziva“ ali nije uvek tako jednostavno, pošto neki modeli pisanja koda mogu da zamagle *stvarno* mesto pozivanja.

Važno je imati na umu *stablo pozivanja* (engl. *call-stack*) (to je stablo koje čine sve funkcije koje su bile pozvane pre nego što smo stigli do tekućeg trenutka izvršavanja). Mesto pozivanja koje nas zanima je *aktivno pre* izvršavanja tekuće funkcije.

Ovako bismo ilustrovali stablo pozivanja i mesto pozivanja:

```
function baz() {
    // pošto se stablo pozivanja sastoji od: `baz`
    // mesto pozivanja ove funkcije nalazi se u
    // globalnom opsegu vidljivosti

    console.log( "baz" );
    bar(); // <-- mesto pozivanja funkcije `bar`
}

function bar() {
    // pošto se stablo pozivanja sastoji od: `baz` -> `bar`
    // mesto pozivanja ove funkcije nalazi se u `baz`

    console.log( "bar" );
    foo(); // <-- mesto pozivanja funkcije `foo`
}
```

```
function foo() {
    // pošto se stablo pozivanja sastoji od: `baz` -> `bar` -> `foo`
    // mesto pozivanja ove funkcije nalazi se u `bar`

    console.log( "foo" );
}

baz(); // <-- mesto pozivanja funkcije `baz`
```

Kada analizirate kôd, imajte na umu da treba pronaći stvarno mesto pozivanja (na stablu pozivanja), zato što je to jedino što je bitno za povezivanje `this`.

Stablo pozivanja možete vizuelizovati u svom umu ako proučite redosled u lancu pozivanja funkcija, kao što smo to uradili u prethodnom primeru koda. Ali to je sporo i podložno greškama. Drugi način da vidite stablo pozivanja jeste da iskoristite alatku za otkrivanje grešaka u svom čitaču veba. Većina savremenih čitača veba namenjenih upotrebi na stonim računarima ima ugrađene razvojne alatke među kojima je i JS dibager. U prethodnom primeru koda, pomoću tih alatki možete postaviti prekidnu tačku u prvi red funkcije `foo()`, ili samo umetnite u prvi red iskaz `debugger;`. Kada izvršavate stranicu, dibager će se zaustaviti na tom mestu i prikazati listu svih funkcija koje su bile pozvane do tog reda, što će biti stablo pozivanja. Znači, ako pokušavate da otkrijete šta predstavlja `this`, upotrebite alatke za otkrivanje grešaka, a zatim pogledajte drugu stavku odozgo – tako ćete saznati stvarno mesto pozivanja koje ste tražili.

Samo po pravilu

Sada ćemo videti *kako* mesto pozivanja određuje na šta će `this` upućivati tokom izvršavanja funkcije.

Potrebno je da ispitate mesto pozivanja i utvrdite koje od četiri pravila važi. Prvo ćemo pojedinačno objasniti svako od ta četiri pravila, a onda ćemo ilustrovati njihov redosled prioriteta, u slučaju da bi se na dato mesto pozivanja *moglo* primeniti više pravila istovremeno.

Podrazumevano povezivanje

Prvo pravilo koje ćemo razmotriti potiče od najuobičajenijeg slučaja pozivanja funkcije: samostalni poziv funkcije. To pravilo za `this` možete zamisliti kao podrazumevano pravilo koje se primenjuje kada nijedno drugo pravilo nije primenljivo.

Razmotrite sledeći kôd:

```
function foo() {
     console.log( this.a );
}

var a = 2;

foo(); // 2
```

Prvo na šta treba da obratite pažnju, ako još niste, jeste to da su promenljive koje su deklarisane u globalnom opsegu vidljivosti, kao što je `var a = 2`, sinonimi za svojstva globalnog objekta istog imena. To nisu kopije jednih u druge – one *jesu* i jedno i drugo. Zamislite ih kao dve strane istog novčića.

Drugo, vidimo da kada se pozove funkcija `foo()`, u njoj se `this.a` razrešava u globalnu promenljivu `a`. Zašto? Zato što se u ovom slučaju, na mestu pozivanja funkcije primenjuje *podrazumevano povezivanje* za `this`, pa identifikator `this` predstavlja globalni objekat.

Kako znamo da se u ovom slučaju primenjuje pravilo *podrazumevanog povezivanja*? Ispitaćemo mesto pozivanja da bismo utvrdili kako se poziva funkcija `foo()`. U navedenom primeru, `foo()` se poziva u obliku obične i nedekorisane reference same funkcije. Pošto se u tom slučaju ne primenjuje nijedno drugo pravilo, važi *podrazumevano povezivanje*.

Ako je uključen striktni režim (tj. važi `strict mode`), globalni objekat nije upotrebljiv za *podrazumevano povezivanje*, pa zato `this` postaje objekat `undefined`:

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` je `undefined`
```

Suptilan ali važan detalj: uprkos tome što se sva pravila za povezivanje `this` zasnivaju na mestu pozivanja funkcije, globalni objekat je upotrebljiv za *podrazumevano povezivanje* samo ako za izvršavanje sadržaja `foo()` nije uključen striktni režim; za samo mesto pozivanja funkcije `foo()`, nebitno je da li je striktni režim uključen ili nije:

```
function foo() {
    console.log( this.a );
}

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```

Namerno mešanje uključenog i isključenog striktnog režima u kodu uglavnom se ne preporučuje. Trebalo bi da striktni režim važi za ceo program ili ni za jedan njegov deo. Međutim, budući da ponekad u program uključujete biblioteku iz trećeg izvora koja radi u drugačijem režimu od vašeg koda, morate voditi računa i o tim suptilnim detaljima kompatibilnosti.

Implicitno povezivanje

Drugo pravilo koje treba uzeti u obzir jeste da li mesto pozivanja ima objekat konteksta, koji se naziva i vlasnički ili objekat okruženja, mada ti alternativni nazivi navode na donekle pogrešan zaključak.

Razmotrite sledeće:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

Prvo, obratite pažnju na oblik u kojem je funkcija `foo()` deklarisana a zatim dodata kao referenca svojstvu objekta `obj`. Bez obzira na to da li je funkcija `foo()` inicijalno deklarisana *unutar* objekta `obj`, ili mu je naknadno dodata kao referenca (kao u navedenom primeru koda), ta funkcija nije ni u kom slučaju „vlasništvo“ objekta `obj`, niti je u njemu „sadržana“.

Međutim, pošto se na mestu pozivanja kontekst objekta `obj` *koristi* za referenciranje funkcije, u trenutku pozivanja funkcije *mogli biste* reći da je objekat `obj` „vlasnik“ reference funkcije, tj. da je „sadrži“.

Kako god da nazovete taj model, na mestu pozivanja funkcije `foo()` prethodi joj referenca na objekat `obj`. Kada se za referenciranje funkcije koristi kontekstni objekat, pravilo *implicitnog povezivanja* nalaže da se *taj* objekat povezuje sa `this` dok se funkcija izvršava. Pošto `obj` jeste `this` tokom izvršavanja funkcije `foo()`, `this.a` je sinonim za `obj.a`.

Na mestu pozivanja važan je samo najviši/poslednji nivo lanca referenciranja svojstava. Na primer:

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

Gubljenje implicitno uspostavljene veze

Jedan od najuobičajenih razloga za nerviranje čiji je izvor povezivanje identifikatora `this`, pojavljuje se kada neka *implicitno povezana* funkcija izgubi tu vezu, što obično znači automatsko prelaženje na *podrazumevano povezivanje* na globalni objekat ili objekat `undefined` – zavisno od toga da li je isključen striktni režim.

Razmotrite sledeće:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // referenca/alijas funkcije!

var a = "uh, globalni"; // `a` je svojstvo globalnog objekta

bar(); // "uh, globalni"
```

Uprkos tome što izgleda kao da je `bar` referenca na `obj.foo`, to je zapravo samo još jedna referenca na samu funkciju `foo`. Osim toga, bitno je jedino šta se događa na mestu pozivanja, a tamo imamo `bar()`, što je običan i nedekorisan poziv funkcije, pa se zato primenjuje *podrazumevano povezivanje*.

Suptilniji, češći i neočekivaniji slučaj nastaje kada pozvanoj funkciji prosleđujemo povratnu funkciju:

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` je samo još jedna referenca na `foo`

    fn(); // <-- mesto pozivanja!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "uh, globalni"; // `a` je svojstvo globalnog objekta

doFoo( obj.foo ); // "uh, globalni"
```

Prosleđivanje parametara je samo implicitno dodeljivanje vrednosti, a pošto prosleđujemo funkciju, to je implicitno dodeljivanje reference, pa je zato konačni rezultat isti kao u prethodnom primeru.

Šta ako funkcija kojoj prosleđujete povratnu funkciju nije vaša, nego je ugrađena u jezik? Nema razlike, rezultat je isti:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = "uh, globalni"; // `a` je svojstvo globalnog objekta

setTimeout( obj.foo, 100 ); // "uh, globalni"
```

Zamislite da je u JavaScript okruženje ugrađena sledeća gruba i teorijska pseudoimplementacija funkcije `setTimeout()`:

```
function setTimeout(fn,interval) {
    // sačekati (nekako) `interval` milisekundi
    // a onda pozvati funkciju fn
    fn(); // <-- mesto pozivanja!
}
```

Prilično je uobičajeno da funkcije koje prosleđujete kao povratne funkcije *izgube* povezivanje svog `this`, kao što smo upravo videli. Još jedan način na koji nas `this` može iznenaditi jeste kada funkcija kojoj smo prosledili provratnu funkciju namerno menja `this` u tom pozivu. Funkcije za obradu događaja u popularnim JavaScript bibliotekama prilično „vole“ da menjaju `this` tako da upućuje, na primer, na DOM element koji je izvor događaja. Mada to ponekad može biti korisno, u drugim slučajevima vas može baš razbesneti. Nažalost, te alatke vam retko kad pružaju mogućnost da sami birate.

U svakom slučaju, `this` je neočekivano izmenjen, a pošto vi nemate zaista kontrolu nad time kako će se izvršiti referenca na vašu povratnu funkciju, (još) nemate način da zadate njeno mesto pozivanja kako biste nametnuli povezivanje `this` u obliku koji vama odgovara. U nastavku teksta razmotrićemo način „rešavanja“ tog problema *ispravljanjem* `this`.

Eksplicitno povezivanje

Pri *implicitnom povezivanju*, kao što smo upravo videli, morali smo da izmenimo dati objekat tako da on funkciji prosleđuje referencu na sebe i da zatim referencu na svojstvo objekta upotrebimo za (implicitno) povezivanje `this` s tim objektom.

Ali, šta ako želimo da se – pri pozivanju funkcije – `this` poveže s konkretnim objektom, ali da pritom ne moramo zadavati i referencu na svojstvo tog objekta?

„Sve“ funkcije u jeziku stavljaju na raspolaganje određene metode (kroz svojstvo `[[Prototype]]` objekta funkcije – više informacija o tome naći ćete u nastavku teksta), što može biti korisno za ovu namenu. Konkretno, objekti funkcija imaju metode `call(..)` i `apply(..)`. Tehnički rečeno, radna okruženja JavaScripta ponekad stavljaju na raspolaganje funkcije koje su toliko specijalne (učtivo govoreći!) da ne nude tu funkcionalnost. Ali takve su malobrojne. Većina funkcija koje postoje – a svakako sve funkcije koje ćete vi napisati – imaju metode `call(..)` i `apply(..)`.

Kako te metode rade? Obe prihvataju kao svoj prvi parametar objekat s kojim treba povezati `this`, a zatim pozivaju funkciju s tim zadatim `this`. Budući da direktno zadajete šta želite da `this` bude, to zovemo *eksplicitno povezivanje*.

Razmotrite sledeće

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Pozivanje funkcije `foo` sa *eksplicitnim povezivanjem* u obliku `foo.call(..)`, omogućava da nametnemo da `this` predstavlja objekat `obj`.

Ako umesto objekta prosledite prostu primitivnu vrednost (tipa `string`, `boolean` ili `number`) za povezivanje sa `this`, ta prosta vrednost konvertuje se u objektni oblik (`new String(..)`, `new Boolean(..)`, odnosno `new Number(..)`). To se često naziva „pakovanje“ (engl. *boxing*) vrednosti.

Obe metode – `call(..)` i `apply(..)` – povezuju `this` na isti način. One se *razlikuju* samo po dodatnim parametrima, ali je to zasad nebitno.

Nažalost, *eksplicitno povezivanje* – samo po sebi – još nam ne pruža nikakvo rešenje za ranije opisani problem funkcije koja „gubi“ već uspostavljeno povezivanje `this`, ili to povezivanje nameće neka spoljašnja biblioteka itd.

Čvrsto povezivanje

Postoji varijanta modela *eksplicitnog povezivanja* koja zapravo obavlja posao. Razmotrite sledeće:

```
function foo() {
    console.log( this.a );
}
```

```
var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
 };

bar(); // 2
setTimeout( bar, 100 ); // 2

// čvrsto povezana `bar` više ne dozvoljava menjanje svog `this`
bar.call( window ); // 2
```

Razmotrimo kako radi ova varijanta. Deklarišemo funkciju `bar()` koja, interno i ručno poziva `foo.call(obj)`, što znači da se funkcija `foo` poziva tako da `this` predstavlja promenljivu `obj`. Bez obzira na to kako posle toga pozivate funkciju `bar`, ona će uvek ručno pozivati funkciju `foo` s promenljivom `obj`. Ta veza je i eksplicitna i čvrsta, pa je zato zovemo *čvrsto povezivanje* (engl. *hard binding*).

Najuobičajeniji način umotavanja funkcije u *čvrstu vezu* formira prolaz za sve argumente koje prosledite i povratnu vrednost koju dobijete:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Drugi način da iskoristite ovaj model jeste da napravite višekratno upotrebljivu pomoćnu funkciju:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// jednostavna pomoćna funkcija za povezivanje this
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
```

```
    };
}

var obj = {
    a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Pošto je *čvrsto povezivanje* tako uobičajen model, u ES5 je na raspolaganju u obliku ugrađene metode, `Function.prototype.bind(..)`, koja se koristi na sledeći način:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Metoda `bind(..)` vraća novu funkciju koja je generisana tako da poziva izvornu funkciju s kontekstom za `this` koji zadate.

Pozivanje API-ja s parametrom „context“

Funkcije iz mnogih biblioteka, kao i mnoge nove funkcije ugrađene u jezik JavaScript i radno okruženje, imaju neobavezan parametar, čije je ime najčešće „context“, a svrha mu je da vas poštedi upotrebe metode `bind(..)` kako biste obezbedili da vaša povratna funkcija koristi baš određeni `this`.

Na primer:

```
function foo(el) {
    console.log( el, this.id );
}

var obj = {
    id: "awesome"
};

// pri pozivanju `foo(..)` kao `this` koristi se `obj`
[1, 2, 3].forEach( foo, obj );
// 1 awesome  2 awesome  3 awesome
```

Interno, sve funkcije te vrste gotovo uvek koriste *eksplicitno povezivanje* pomoću `call(..)` ili `apply(..)` da vam uštede trud.

Povezivanje funkcija pomoću operatora new

Četvrto i poslednje pravilo za povezivanje `this` zahteva da se vratimo na veoma često pogrešno mišljenje u vezi sa funkcijama i objektima u JavaScriptu.

U tradicionalnim jezicima orijentisanim na klase, „konstruktori" su specijalne metode pridružene klasama. Kada pomoću operatora `new` pravite instancu klase, poziva se konstruktor te klase. To obično izgleda slično ovome:

```
nekaInstanca = new NekaKlasa(..);
```

JavaScript ima operator `new` i model njegove upotrebe u kodu izgleda isto kao u jezicima orijentisanim na klase; većina programera podrazumeva da JS mehanizam radi nešto slično. Međutim, delovanje operatora `new` u JavaScriptu zapravo *nema nikakve sličnosti* s funkcionalnošću kakvu nude jezici orijentisani na klase.

Prvo ćemo redefinisati šta je „konstruktor" u JavaScriptu. U JS-u, konstruktori su samo funkcije koje su pozvane tako da im prethodi operator `new`. One nisu vezane za klase, niti generišu instance klasa. Nisu čak ni specijalne vrste funkcija. To su samo obične funkcije koje, u suštini, rade malo drugačije zato što su pozvane s operatorom `new`.

Na primer, razmotrite funkciju `Number(..)` kada igra ulogu konstruktora, prema citatu iz specifikacija za ES5.1:

> 15.7.2 Konstruktor Number
>
> Kada se unutar izraza funkcija Number pozove s operatorom `new`, ona postaje konstruktor i inicijalizuje novonapravljeni objekat.

Dakle, manje-više svaka obična funkcija, uključujući i metode objekata, kao što je `Number(..)` (videti poglavlje 11), može se pozvati s operatorom `new` ispred, da bi taj poziv funkcije onda postao *konstruktorski poziv*. To je važna suptilna razlika: ne postoji ništa što bi bilo „konstruktorska funkcija" – postoje samo *konstruktorski pozivi funkcija*.

Kada funkciju pozivate s operatorom `new` ispred, što se inače zove konstruktorski poziv, automatski se obavlja sledeće:

1. Pravi se (konstruiše) potpuno nov objekat.
2. Novokonstruisani objekat se povezuje pomoću svog svojstva `[[Prototype]]`.
3. Novokonstruisani objekat se zadaje kao objekat koji `this` predstavlja u ovom pozivu funkcije.
4. Ako ne vraća vlastiti alternativni objekat, funkcija pozvana s operatorom `new` *automatski* vraća novokonstruisani objekat.

Koraci 1, 3 i 4 odnose se na naše tekuće razmatranje. Korak 2 ćemo zasad preskočiti i vratiti se na njega u poglavlju 13.

Razmotrite sledeći kôd:

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

Pozivanjem funkcije `foo(..)` s operatorom `new` ispred nje, konstruisali smo nov objekat i zadali da je on `this` u ovom pozivu funkcije `foo(..)`. Dakle, upotreba `new` je poslednji način na koji se pri pozivanju funkcije može povezati identifikator `this`. To ćemo nazvati *povezivanje pomoću new*.

Uvek po redosledu

Sada znamo četiri pravila za povezivanje `this` pri pozivanju funkcija. Samo treba da pronađete mesto pozivanja i ispitate ga kako biste utvrdili koje se pravilo primenjuje. Ali, šta ako je na mestu pozivanja moguća primena više pravila istovremeno? Pošto mora postojati određeni redosled prioriteta tih pravila, sada ćemo ilustrovati redosled kojim se ona primenjuju.

Trebalo bi da bude jasno da je *podrazumevano povezivanje* pravilo s najnižim prioritetom od sva četiri. Zato ćemo ga preskočiti.

Šta ima viši prioritet, *implicitno povezivanje* ili *eksplicitno povezivanje*? Ispitajmo šta se događa:

```
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
foo();   // undefined
```

Dakle, *eksplicitno povezivanje* ima viši prioritet od *implicitnog povezivanja*, što znači da bi prvo trebalo da se zapitate da li je primenjeno *eksplicitno povezivanje* pre nego što ispitate da li je primenjeno *implicitno povezivanje*.