POGLAVLJE 17

Izvorni objekti

U poglavljima 15 i 16 pomenuli smo više ugrađenih objekata, koji se obično zovu „izvorni“ objekti (engl. *natives*), kao što su `String` i `Number`. Sada ćemo ih detaljno razmotriti.

Najčešće se koriste sledeći izvorni objekti:

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()` – dodat u ES6!

Kao što vidite, ti izvorni objekti su zapravo ugrađene funkcije.

Ako u JS dolazite iz nekog jezika kao što je Java, JavaScriptov `String()` će vam izgledati kao konstruktor `String(..)` pomoću kojeg ste navikli da konstruišete vrednosti znakovnog tipa. Zato ćete brzo shvatiti da možete raditi ovakve stvari:

```
var s = new String( "Zdravo svete!" );

console.log( s.toString() ); // "Zdravo svete!"
```

Tačno je da se svaka od tih izvornih funkcija može koristiti kao izvorni konstruktor. Ali ono što se tako konstruiše može biti različito od onog što mislite:

```
var a = new String( "abc" );

typeof a;                            // "object" ... nije "String"

a instanceof String;                 // true

Object.prototype.toString.call( a ); // "[object String]"
```

Rezultat generisanja vrednosti pomoću konstruktora (`new String("abc")`) jeste objektni omotač u koji je upakovana primitivna vrednost (`"abc"`).

Važno je to da operator `typeof` pokazuje da ti objekti nemaju vlastite specijalne *tipove*, nego su zapravo samo podtipovi opšteg tipa `object`.

Postojanje objektnog omotača može se prepoznati i pomoću sledećeg iskaza:

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

Rezultat ovog iskaza zavisi od vašeg čitača veba, budući da razvojne konzole mogu slobodno birati najprikladniji oblik u kojem serijalizuju objekat da bi ga prikazale programeru.

U vreme pisanja ove knjige, najnovija verzija čitača Chrome prikazuje nešto slično sledećem: `String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`. Starije verzije Chromea prikazale bi samo nešto nalik sledećem: `String {0: "a", 1: "b", 2: "c"}`. Poslednja verzija Firefoxa sada prikazuje `String ["a","b","c"]`, ali je ranije prikazivala `"abc"` kurzivnom slovima, što je na pritisak miša otvaralo inspektor objekta. Razume se, opisani rezultati su vrlo podložni promenama, a vaše lično iskustvo može biti drugačije.

Suština je to da iskaz `new String("abc")` konstruiše omotački objekat u koji je upakovan znakovni niz `"abc"`, a ne samu primitivnu vrednost `"abc"`.

Interno svojstvo [[Class]]

Vrednosti za koje operator `typeof` pokazuje `"object"` (kao što je niz) imaju dodatno interno svojstvo `[[Class]]` (koje treba posmatrati više kao internu *klas*ifikaciju nego kao nešto što ima veze s klasama u smislu tradicionalnog koda orijentisanog na klase). Tom svojstvu se ne može pristupati direktno, ali se njegova vrednost može utvrditi indirektno, pozivanjem metode `Object.prototype.toString(..)` za samu vrednost. Na primer:

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

Iz tog razloga, za niz u navedenom primeru, vrednost internog svojstva `[[Class]]` je `"Array"`, a za regularni izraz, to je `"RegExp"`. U većini slučajeva, vrednost tog internog svojstva `[[Class]]` odgovara ugrađenom izvornom konstruktoru (videti u nastavku teksta) koji se odnosi na vrednost – ali nije uvek tako.

Šta je s primitivnim vrednostima? Prvo, `null` i `undefined`:

```
Object.prototype.toString.call( null );
// "[object Null]"

Object.prototype.toString.call( undefined );
// "[object Undefined]"
```

Obratite pažnju na to da postoje izvorni konstruktori `Null()` ili `Undefined()`, ali uprkos tome, vrednosti internog svojstva `[[Class]]` koje se prikazuju jesu `"Null"` i `"Undefined"`.

U slučaju drugih prostih primitiva – kao što su `string`, `number` i `boolean` – imamo zapravo drugačije ponašanje, koje se najčešće naziva „pakovanje“ (videti odeljak „Pakovanje primitiva u objektne omotače“ na strani 42):

```
Object.prototype.toString.call( "abc" );
// "[object String]"

Object.prototype.toString.call( 42 );
// "[object Number]"

Object.prototype.toString.call( true );
// "[object Boolean]"
```

U ovom primeru, svaka od tih prostih primitiva automatski se umeće u odgovarajući objektni omotač, što je razlog zbog kojeg se kao vrednosti njihovog internog svojstva `[[Class]]` prikazuju `"String"`, `"Number"` i `"Boolean"`.

Ponašanja metode `toString()` i svojstva `[[Class]]` koja su ovde opisana, malo su se promenila od verzije ES5 do ES6, ali ćemo te detalje razmotriti u šestom delu ove knjige – *ES6 i sledeće verzije*.

Pakovanje primitiva u objektne omotače

Objektni omotači služe vrlo važnoj svrsi. Pošto primitivne vrednosti nemaju ni svojstva ni metode, da biste upotrebili metodu `.length` ili `.toString()` treba vam objektni omotač oko primitivne vrednosti. Srećom, JS će automatski *upakovati* (engl. *box*) (tj. umotati) primitivnu vrednost da bi omogućio takve pristupe:

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

Dakle, ako nameravate da u svojim znakovnim nizovima često pristupate tim svojstvima/metodama – kao što je, na primer, uslov `i < a.length` u petlji `for` – možda bi vam izgledalo logično da od samog početka radite isključivo s objektnim oblikom vrednosti, pa zato nema potrebe da ga JS mašina implicitno pravi za vas.

Ali, ispostavlja se da je to loša ideja. Performanse čitača veba odavno su optimizovane za vrlo česte slučajeve kao što je `.length`, što znači da će vaš program *zapravo raditi sporije* ako pokušate da ga „unapred optimizujete“ tako što direktno koristite objektni oblik (koji se ne nalazi na optimizovanoj putanji).

Uglavnom, nema pravog razloga da se direktno koristi objektni oblik. Bolje je da prepustite mašini jezika da implicitno pakuje vrednosti na mestima gde je to neophodno. Drugim rečima, nemojte nikad raditi stvari kao što su `new String("abc")`, `new Number(42)` itd. – uvek radije radite s literalnim primitivnim vrednostima `"abc"` i `42`.

Zamke upotrebe objektnih omotača

Postoje čak i zamke koje se pojavljuju pri direktnoj upotrebi omotača, o kojima bi trebalo da vodite računa ako se ipak *namerno* opredelite za njihovu upotrebu.

Na primer, razmotrite sledeće upakovane vrednosti tipa `Boolean`:

```
var a = new Boolean( false );

if (!a) {
    console.log("Ups" ); // ovo se nikada ne izvršava
}
```

Problem je to što smo vrednost `false` upakovali u objekat, ali pošto se svi objekti po svojoj prirodi ponašaju kao `true` (videti poglavlje 18), ispitivanje objekta daje suprotan rezultat od ispitivanja njegove interne vrednosti `false`, što je sasvim suprotno ponašanje od očekivanog.

Ako želite da ručno upakujete primitivnu vrednost, upotrebite funkciju `Object(..)` (a ne rezervisanu reč `new`):

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

Ponavljam, direktna upotreba objektnog omotača umesto same vrednosti (kao `b` i `c` u navedenom primeru) uglavnom se ne preporučuje, ali može se dogoditi da naiđete na određene retke slučajeve gde to može biti korisno.

Raspakivanje

Ako imate objektni omotač i želite da iz njega izdvojite njegovu internu primitivnu vrednost, upotrebite metodu `valueOf()`:

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

Raspakivanje se može obaviti i implicitno, ako vrednost unutar objektnog omotača upotrebite na način koji zahteva primitivnu vrednost. Taj postupak (konverzija tipa) detaljnije je opisan u poglavlju 18, ali ukratko rečeno:

```
var a = new String( "abc" );
var b = a + ""; // `b` sadrži raspakovanu primitivnu vrednost "abc"

typeof a;       // "object"
typeof b;       // "string"
```

Izvorne funkcije kao konstruktori

Za vrednosti koje su tipa `array`, `object`, `function` ili regularni izrazi, gotovo je uvek bolje rešenje da ih konstruišete u literalnom obliku, budući da u oba slučaja dobijate objekat, a ne skalarnu vrednost.

Kao što smo već videli u prethodnom delu poglavlja u slučaju drugih izvornih funkcija, te konstruktorske oblike treba uglavnom izbegavati – osim ako ste zaista sigurni da vam trebaju – prvenstveno zato što uvode izuzetke i zamke s kojima verovatno *ne želite* da imate posla.

Konstruktor Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

Ispred konstruktora `Array(..)` ne morate zadati rezervisanu reč `new`. Ako je izostavite, ponašaće se isto kao da ste je zadali. Zato `Array(1,2,3)` daje isti rezultat kao `new Array(1,2,3)`.

Konstruktor `Array` ima specijalan oblik u kojem ako mu prosledite samo jedan argument tipa `number`, umesto da tu vrednost obrađuje kao *sadržaj* niza, tumači je kao veličinu za koju „unapred priprema“ niz (ili tako nekako).

To je vrlo loša ideja. Prvo, taj oblik možete zadati nenamerno jer se na to lako zaboravlja.

Što je još važnije, ne postoji način da unapred pripremite niz zadate veličine. Umesto toga, ono što zaista dobijate je prazan niz, ali čije svojstvo `length` ima numeričku vrednost koju ste zadali.

Niz čiji elementi nemaju nikakve eksplicitne vrednosti, ali čije je svojstvo `length` takvo da *navodi na pomisao* da ti elementi postoje, jeste vrsta čudne i egzotične strukture za podatke u JavaScriptu, koja se ponaša vrlo neobično i zbunjujuće. Mogućnost stvaranja takvih vrednosti potiče isključivo od danas zastarelih i istorijskih funkcionalnosti („objekti nalik nizovima“ kao što je objekat `arguments`).

Niz koji sadrži barem jedan „prazan element“ često se naziva „proređen niz“.

Situaciju ne poboljšava to što je ovo još jedan primer gde razvojne konzole u čitaču veba mogu slobodno određivati oblik u kojem će predstaviti takav objekat, što dodatno zbunjuje.

Na primer:

```
var a = new Array( 3 );

a.length; // 3
a;
```

Oblik u kojem se `a` serijalizuje u čitaču Chrome (u vreme pisanja ove knjige) izgleda ovako: `[ undefined x 3 ]`. To je zaista nesrećan izbor jer podrazumeva da taj niz ima tri elementa čije su vrednosti `undefined`, dok u stvari ti elementi ni ne postoje (takozvani „prazan niz“ – još jedno loše ime!).

Da biste shvatili razliku, pokušajte sledeće:

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```

Kao što vidite u slučaju promenljive `c` u ovom primeru, nakon definisanja niza mogu se pojaviti prazni elementi. Kada svojstvo `length` niza izmenite tako da ne pokazuje tačan broj vrednosti u zaista definisanim elementima, implicitno uvodite prazne elemente. U stvari, u prethodnom primeru možete pozvati čak i metodu `delete b[1]`, što uvodi prazan element usred niza `b`.

Promenljiva `b` (zasad u čitaču Chrome) serijalizuje se u oblik `[ undefined, undefined, undefined ]`, za razliku od oblika `[ undefined x 3 ]` za `a` i `c`. Zbunjeni? Da, kao što bi i svako bio.

Što je još gore, u vreme pisanja ove knjige, Firefox prikazuje `[ , , , ]` za `a` i `c`. Da li ste shvatili šta toliko zbunjuje? Pogledajte pažljivije. Tri zareza znače da u nizu postoje četiri elementa, a ne tri kao što bismo očekivali.

Otkud sad to!? Firefox dodaje na kraj svog oblika serijalizovanja jedan zarez jer su – od specifikacije za ES5 – u listama (vrednosti u nizu, svojstava itd.) dozvoljeni završni zarezi (pa se zato izostavljaju i zanemaruju). Znači, ako u svom programu ili na konzoli upišete vrednost `[ , , , ]`, dobićete zapravo vrednost koja je `[ , , ]` (tj. niz s tri prazna elementa). Takvo rešenje – mada zbunjuje ako čitate sadržaj razvojne konzole – brani se time da čini tačnim ponašanje pri kopiranju i umetanju.

Ako u ovom trenutku odmahujete glavom i prevrćete očima, niste jedini! Ali tako stoje stvari.

Izgleda kao da u ovakvim slučajevima Firefox menja svoj rezultat u `Array [ <3 empty slots> ]`, što je svakako veliki korak napred u poređenju sa `[ , , , ]`.

Nažalost, postaje sve gore. Gore od toga da se samo prikazuju u zbunjujućem obliku na razvojnoj konzoli, `a` i `b` iz prethodnog primera koda zapravo se u nekim slučajevima ponašaju isto, a u drugim slučajevima različito:

```
a.join( "-" ); // "--"
b.join( "-" ); // "--"

a.map(function(v,i){ return i; }); // [ undefined x 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

Ajoj.

Metoda `a.map(..)` *greši* iz sledećeg razloga: pošto elementi niza zapravo ne postoje, metoda `map(..)` nema ništa što bi mogla da obrađuje u iteraciji. Metoda `join(..)` radi drugačije. Možemo je zamisliti kao da je implementirana u sledećem obliku:

```
function fakeJoin(arr,connector) {
    var str = "";
    for (var i = 0; i < arr.length; i++) {
        if (i > 0) {
            str += connector;
        }
        if (arr[i] !== undefined) {
            str += arr[i];
        }
    }
    return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"
```

Kao što vidite, metoda `join(..)` radi tako što *pretpostavlja* da elementi postoje i obrađuje ih u petlji koja se izvršava `length` puta. Evo šta metoda `map(..)` interno radi: pošto (verovatno) ne polazi od iste pretpostavke, njen rezultat za čudan niz s „praznim elementima“ neočekivan je i verovatno će prouzrokovati greške.

Dakle, ako *zaista* želite da napravite niz čiji su elementi vrednosti `undefined` (a ne „prazni elementi“), kako biste to uradili (osim ručno)?

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

Zbunjeni? Da. Evo kako to otprilike radi.

Metoda `apply(..)` na raspolaganju je u svim funkcijama i poziva funkciju za koju je upotrebite, ali na poseban način.

Njen prvi argument je objekat na koji upućuje `this` (opisan u trećem delu ove knjige – *Identifikator this i prototipovi objekata*), za koji, pošto nam je u ovom slučaju nebitan, zadajemo `null`. Trebalo bi da drugi argument bude niz (ili nešto što se *ponaša kao niz* – tj. „objekat nalik nizu“). Elementi tog „niza“ raspodeljuju se kao argumenti funkcije koja se poziva.

Dakle, `Array.apply(..)` poziva funkciju `Array(..)` kojoj kao argumente prosleđuje raspodeljene vrednosti (iz objektne vrednosti `{ length: 3 }`).

Unutar metode `apply(..)`, možemo pretpostaviti da postoji još jedna petlja `for` (otprilike kao u metodi `join(..)` iz prethodnog dela) koja se izvršava od `0` do – ali ne i uključujući – `length` puta (`3` u našem primeru).

Za svaki indeks, metoda učitava ključ iz objekta. Tako da ako objekat koji prosledite na mestu parametra za niz ima unutar funkcije `apply(..)` interno ime `arr`, svojstvima tog objekta pristupaće se u obliku `arr[0]`, `arr[1]` i `arr[2]`. Razume se, pošto nijedno od tih svojstava ne postoji u objektu `{ length: 3 }`, svako od ta tri pristupanja svojstvu vraća vrednost `undefined`.

Drugim rečima, funkcija `Array(..)` u suštini se poziva u sledećem obliku: `Array(undefined,undefined,undefined)`, što je razlog zbog kojeg dobijamo niz popunjen vrednostima `undefined`, a ne samo one (nezgodne) prazne elemente.

Iako je `Array.apply( null, { length: 3 } )` neobičan i opširan način za izradu niza čiji su elementi vrednosti `undefined`, to je *znatno* bolji i pouzdaniji način od pucanja sebi u nogu s praznim elementima zbog `Array(3)`.

Zaključak: *nikad, ni u kom slučaju*, nemojte namerno praviti ni koristiti egzotične nizove s praznim elementima. To se ne radi. Oni su zlo.

Konstruktori Object(..), Function(..) i RegExp(..)

Konstruktori `Object(..)`/`Function(..)`/`RegExp(..)` najčešće su neobavezni (pa ih stoga treba uglavnom izbegavati, osim kada su zaista neophodni):

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; }
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

Praktično ne postoji razlog za upotrebu konstruktorskog oblika `new Object()`, a naročito zato što vas to primorava da svojstva ručno dodajete objektu jedno po jedno, umesto da ih sve zadate u jednom iskazu kao u literalnom obliku definicije objekta.

Konstruktor `Function` je koristan samo u izuzetno retkom slučaju kada parametre ili telo funkcije treba da definišete dinamički. Nemojte posmatrati `Function(..)` samo kao alternativni oblik za `eval(..)`. Gotovo nikad vam neće trebati da funkcije definišete dinamički na taj način.

Snažno se preporučuje da se regularni izrazi definišu u literalnom obliku (`/^a*b+/g`) – ne samo zbog jednostavnije sintakse, nego i zbog boljih performansi: JS mašina ih unapred kompajlira i smešta u keš, pre izvršavanja koda. Za razliku od drugih konstruktora koje smo dosad razmatrali, `RegExp(..)` pruža određenu razumnu korist: za dinamičko definisanje šablona regularnog izraza:

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

Pošto se ova vrsta slučaja sasvim normalno ponekad pojavljuje u JS programima, moraćete da upotrebite oblik `new RegExp("pattern","flags")`.

Konstruktori Date(..) i Error(..)

Izvorni konstruktori `Date(..)` i `Error(..)` znatno su korisniji od drugih izvornih konstruktora, zato što ni za jedan od tih objekata ne postoji literalni oblik.

Da biste konstruisali objektnu vrednost koja predstavlja datum, morate upotrebiti oblik `new Date()`. Konstruktor `Date(..)` prihvata opcione argumente pomoću kojih se zadaju datum/vreme, ali ako ih izostavite, uzima se tekući datum/vreme.

Najčešći razlog da konstruišete datumski objekat jeste da dobijete tekuću vrednost Unix vremenske oznake (što je celobrjna vrednost koja predstavlja broj sekundi protekao od 1. januara 1970. godine). To možete postići pomoću metode `getTime()` instance datumskog objekta.

Još lakši način je da samo pozovete statičku pomoćnu funkciju definisanu u ES5: `Date.now()`. Polifil za okruženja starija od ES5 vrlo je jednostavan:

```
if (!Date.now) {
    Date.now = function(){
        return (new Date()).getTime();
    };
}
```

Ako konstruktor `Date()` pozovete bez `new`, dobijate tekstualni oblik tekućeg datuma/vremena. Tačan oblik tog podatka nije zadat u specifikacijama jezika, mada se mnogi čitači veba slažu oko nečega sličnog sledećem: `"Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)"`.

Konstruktor `Error(..)` (vrlo slično prethodnom konstruktoru `Array()`) ponaša se isto, bez obzira na to da li uz njega zadate rezervisanu reč `new`.

Glavni razlog zbog kojeg biste napravili objekat koji predstavlja grešku jeste da biste u taj objekat preuzeli sadržaj tekućeg stabla (steka) pozivanja funkcija (koje je u većini JS mašina na raspolaganju kao vrednost svojstva samo za čitanje `.stack` nakon konstruisanja objekta). Taj kontekst stabla pozivanja uključuje stablo pozivanja funkcija i redni broj reda u kojem je konstruisan objekat greške, što znatno olakšava otkrivanje i otklanjanje grešaka.

Uobičajeno je da se takav objekat greške pravi pomoću operatora `throw`:

```
function foo(x) {
    if (!x) {
        throw new Error( "x nije zadato" );
    }
    // ..
}
```

Instance objekata grešaka uglavnom imaju barem svojstvo `message`, a ponekad i druga svojstva (koja bi trebalo da tumačite kao samo za čitanje), kao što je `type`. Međutim, osim ispitivanja pomenutog svojstva `stack`, uglavnom je najbolje da pozovete metodu `toString()` objekta greške (eksplicitno, ili implicitno kroz konverziju tipa – videti poglavlje 18) da biste dobili čitljivije formatiranu poruku o grešci.

Tehnički govoreći, osim opšteg izvornog konstruktora `Error(..)`, postoji i više drugih izvornih konstruktora koji su uže specijalizovani za pojedine vrste grešaka: `EvalError(..)`, `RangeError(..)`, `ReferenceError(..)`, `SyntaxError(..)`, `TypeError(..)` i `URIError(..)`. Ali se ti konstruktori za specifične greške vrlo retko ručno koriste. Oni se automatski pozivaju ako se u programu javi stvarni izuzetak (kao što je referenciranje nedeklarisane promenljive, što izaziva grešku `ReferenceError`).

Konstruktor Symbol(..)

U verziji ES6 uvedena je novina nazvana „Symbol“. Vrednosti tipa Symbol su specijalne „jedinstvene“ (što nije striktno garantovano!) vrednosti koje se mogu koristiti kao imena svojstava objekata bez bojazni od dupliranja. One su prvenstveno namenjene za specijalna ugrađena ponašanja ES6 konstrukata, ali možete definisati i vlastite simbole.

Simboli se mogu koristiti za imena svojstava, ali konkretno ime simbola ne možete videti, niti mu pristupati u programu, a ni iz razvojne konzole. Ako pokušate da prikažete ime simbola u razvojnoj konzoli, prikazaće se, na primer, `Symbol(Symbol.create)` – ili nešto slično.

U ES6 postoji više unapred definisanih simbola, kojima se pristupa kao statičkim svojstvima funkcijskog objekta `Symbol`, kao što su `Symbol.create`, `Symbol.iterator` itd. Da biste ih upotrebili, zadajte nešto poput:

```
obj[Symbol.iterator] = function(){ /*..*/ };
```