JavaScript ES6

Autori:
Milan Segedinac
Goran Savić

1. ES6

ECMA Script, skraćeno ES, je specifikacija skripting jezika standardizovana od strane ECMA International[1] nastao sa ciljem da se standardizuje JavaScript i da se time omoguće različite nezavisne implementacije tog jezika. Rane verzije ovog jezika, ES1 iz 1997. godine i ES2 iz 1998. nisu bile široko korišćene. Prva široko prihvaćena verzija bila je ES3 specifikacija iz 1999. godine u kojoj su uvedeni regularni izrazi, pojednostavljeno rukovanje stringovima, uvedeno rukovanje izuzecima i nove kontrolne strukture. Ova verzija jezika i danas se često koristi, posebno kada je potrebno da aplikacije budu podržane i u starijim pregledačima - IE 6-8 i Android 2.x mobilni browser. Verzija ES4 nikada nije prihvaćena zbog neslaganja oko kompleksnosti jezika. Vezija ES5 nastala je 2009. godine i donela je svega nekoliko značajnih novina u odnosu na ES3 - get i set metode, bolju podršku za JSON i bolju podršku za refleksiju atributa objekata. Ova verzija jezika je danas najkorišćenija, pre svega zbog podrške starijim pregledačima. Za razliku od ES5, verzija ES6 iz 2015. godine donela je brojne nove mogućnosti i u ovoj lekciji ćemo se baviti nekim od najvažnijih.

Obzirom da su, u vreme pisanja ovih materijala, još uvek u upotrebi brojni pregledači u kojima ES6 nije podržan ili je pak samo delimično podržan, programski kod napisan u ES6 se najčešće *transpilira* - prevodi korišćenjem posebnog programa kao što je, na primer, Babel[2] - u ES5 programski kod. Transpiliranje je proces prevođenja izvornog koda iz jednog programskog jezika u drugi. Naravno, činjenica da se bilo koji program napisan u ES6 može prevesti u ES5 znači da su nova svojstva uvedena u ES6 samo *sintaktički šećer* za mogućnosti ES5. U slučajevima u kojima je to ilustrativno pogledaćemo i kako programski kod preveden u ES5 izgleda.

Opseg vidljivosti

U ES5 opseg vidljivosti lokalnih varijabli uvek je bila funkcija. Sve lokalne varijable su se deklarisale rezervisanom rečju `var`. U ES6 moguće je i deklarisati varijablu čiji opseg viljivosti je blok. U tom slučaju, za deklarisanje varijable se koristi ključna reč `let`. Primer je dat listingom ispod.

```
var a = 2;

{
      let a = 3;
      console.log( a ); // 3
}
console.log( a ); // 2
```

Listing - `let` deklaracija varijable

Van bloka koda varijabla je deklarisana sa `var` što znači da je njen opseg vidljivosti funkcija. U bloku je varijabla istog naziva deklarisana sa `let`, što znači da je njen opseg vidljivosti taj blok.

---

[1] http://www.ecma-international.org/

[2] https://babeljs.io/

Pogledajmo na listingu ispod kako programski kod izgleda kada se prevede u ES5[3].

```
var a = 2;


{
  var _a = 3;
  console.log(_a); // 3
}

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

Listing - `let` deklaracija prevedena u ES5

Na listingu vidimo da je let deklaracija realizovana uvođenjem nove varijable (`_a`) koja je deklarisana pomoću `var`.

U narednom primeru deklaracija varijabli u programskom jeziku JavaScript napravićemo listu u koju ćemo dodavati funkcije. Svaka funkcija će ispisati u konzolu tekst „`pozvana je funkcija na indeksu "+<indeksUListi>`. Naivna implementacija data je u listingu ispod.

```
var listaFunkcija = [];

for (var i = 0; i < 5; i++) {
      listaFunkcija.push(function () {
            console.log('pozvana je funkcija na indeksu',i-1);
      });
}
listaFunkcija[2]();
```

Listing - `var` deklaracija u petlji

Napravili smo jednu praznu listu. Zatim smo u for petlji dodali pet funkcija u tu listu, od koji svaka ispisuje verenost varijable `i`. Kada pozovemo funkciju na indesku 2 (`listaFunkcija[2]();`), dobijamo ispis „`pozvana je funkcija na indeksu 4`“. Zašto programski kod nije rezultovao ispisom „`pozvana je funkcija na indeksu 2`“?

Obratimo pažnju na deklaraciju varijable `i`. Varijabla i je deklarisana pomoću `var`, što znači da je vidljiva u celoj funkciji u kojoj se nalazi petlja. Svaka izmena (uvećanje varijable `i` u zaglavlju for petlje) odraziće se na istu varijablu. Na kraju for petlje varijabla `i` će imati vrednost 4 i bez obzira koju funkciju iz liste da pozovemo ispisivaće se ista vrednost.

Kako bismo mogli da rešimo ovaj problem? Trebala bi nam lokalna kopija varijable `i` koja je dostupna samo prilikom ispisa u konzolu. To bismo mogli da postignemo tako što ćemo

---

[3] U trenutku pisanja ovih materijala prevođenje pomoću babela radi se tako što se globalno instalira npm paket babel-cli, lokalno instalira npm paket babel-presets-es2015, a zatim izvrši komanda `babel --presets es2015 primer1-uc.js -o primer1.js` . Više detalja je dostupno na https://babeljs.io/ .

dodavanje u listu funkcije za ispis u konzolu obuhvatiti funkcijom kojoj `i` prosledimo kao parametar i odmah izvršiti tu funkciju. Programski kod je dat listingom ispod.

```
var listaFunkcija = [];

for (var i = 0; i < 5; i++) {
      (function (x) {
            listaFunkcija.push(function () {
                  console.log('pozvana je funkcija na indeksu',x-1);
            });
      }(i));
}
listaFunkcija[2]();
```

Listing - `var` deklaracija u petlji i lokalna kopija varijable

U for petlji smo napravili funkciju koja prima jedan parametar, `x`. Pošto nam neće trebati kasnije, funkciji nismo dodelili naziv. U toj funkciji smo realizovali dodavanje u listu i odmah funkciju pozvali prosledivši joj `i` kao argument. Parametar `x` u telu funkcije ima lokalnu kopiju varijable `i` iz svake iteracije. Ovaj šablon - kreirati neimenovanu funkciju i odmah je izvršiti - je veoma čest u programskom jeziku JavaScript jer nam omogućuje da jednostavno dobijemo izolovanost koda i lokalne kopije vrednosti varijabli. Taj šablon se naziva *immediately-invoked function expression*, (*iife*, izgovara se ifi).

Deklaracija varijable pomoću `let` omogućava nam da ovaj problem jednostavnije rešimo. Da je varijabli i bila deklarisana sa `let`, umesto sa `var`, u svakoj iteraciji bi se napravila nova kopija varijable `i`, što bi bilo rešenje problema. Pogledajmo programski kod nakon transpiliranja na listingu ispod.

```
var listaFunkcija = [];


var _loop = function _loop(i) {
      listaFunkcija.push(function () {
            console.log('pozvana je funkcija na indeksu', i - 1);
      });
};

for (var i = 0; i < 5; i++) {
      _loop(i);
}
listaFunkcija[2]();
```

Listing - `let` deklaracija u zaglavlju petlje

Vidimo da je iskorišćen isti pristup koji smo i mi iskoristili da kreiranje lokalne kopije varijable: napravljena je funkcija u koju je varijabla „zatvorena“. Jedina razlika je u tome što smo mi iskoristili neimenovanu funkciju, a u transpiliranom kodu funkcija nije neimenovana, već se zove `_loop`.

Konstante

Konstante su blokovske varijable kojima nije moguće dodeliti novu vrednost. Deklarišu se ključnom rečju `const`. Vodite računa da, ako je konstanta mutabilna, na primer ako je niz

ili ako je objekat, moguće je promeniti njen sadržaj. Primer korišćenja konstanti dat je listingom ispod.

```
{

  const a = [1,2,3];
  a.push( 4 );
  console.log( a ); // [1,2,3,4]
  a = 42; // TypeError!
}
```

Listing - konstante

Prilikom transpiliranje navedenog progmraskog koda biće prijavljena greška u liniji u kojoj konstanti a pokušavamo da dodelimo novu vrednost.

Spread/rest operator

Spread/rest operator se zapisuje kao . . . (tri tačke) i omogućuje jednostavno okupljanje vrednosti u niz i razvijanje niza u vrednosti. Najčešće se koristi u pozivu funkcija. Primer je dat listingom ispod.

```
function foo(x,y,z) {

      console.log( x, y, z );
}
foo( ...[1,2,3] ); // 1 2 3
//foo( 1,2,3 ); // 1 2 3

function bar(x, y, ...z) {
console.log( x, y, z );
}
bar( 1, 2, 3, 4, 5 );
//bar( 1, 2, [3,4,5])
```

Listing - spread/rest operator

Default vrednosti parametara funkcije

Od verzije ES6 moguće je zadati default vrednosti parametara funkcije. U zaglavlju deklaracije funkcije parametrima se dodele vrednosti i, ako nisu prosleđeni argumenti za te parametre, navedene default vrednosti će biti iskorišćene. Primer je dati u listingu ispod.

```
function foo(x = 11, y = 31) {

  console.log( x + y );
}
```

Listing - default vrednosti parametara funkcije

Destrukturiranje (strukturirana dodela vrednosti)

Strukturirana dodela vrednosti je JavaScript izraz koji nam omogućuje da vrednosti iz niza ili atribute objekta dodelimo nezavisnim varijablama (da ih „raspakujemo“ u različite varijable). Listingom ispod dat je primer destrutkuriranja niza u varijable.

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

console.log(a, b, c);
```

Lsting - destrukturiranje niza

Varijable `a`, `b` i `c` su dobile prve tri vrednosti iz niza, odnosno varijabla `a` je dobila vrednost `1`, `b` vrednost `2`, a `c` vrednost `3`. Obratite pažnju da su preostale vrednosti ignorisane.

Primer destrukturiranja objekta dat je listingom ispod.

```
var {x:x1,z:z1} = {x:1,y:2,z:3};

console.log(x1, z1);
```

Listing - destrukturiranje objekta

Varijabla `x1` dobila je vrednost atributa `x` objekta, odnosno dobila je vrednost `1`, a varijabla `z1` dobila je vrednost atributa `z` objekta, odnosno vrednost `3`.

Prilikom destrukturiranja nije neophodno da se dodele svi atributi objekta niti sve vrednosti niza. Videli smo da možemo ignoristati atribute objekta prosto tako što ih nećemo navesti u dodeli i poslednje vrednosti niza tako što ćemo navesti manje varijabli nego što niz ima elemenata. Ali šta bi se desilo da hoćemo da ignorišemo početne vrednosti niza ili središnje vrednosti niza? Primer je dat listingom ispod.

```
var [,,c,,e] = [1,2,3,4,5];

console.log(c,e);
```

Listing - ignorisanje vrednosti iz niza prilikom destrukturiranja

U navedenom primeru prve dve vrednosti niza su ignorisane, zatim je treća vrednost (`3`) dodeljena varijabli `c`, pa je četvrta vrednost ignorisana i na kraju peta vrednost (`5`) dodeljena varijabli `e`.

Destrukturiranje se može koristiti u kombinaciji sa spread/rest operatorom. Primer je dat listingom ispod.

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

console.log(c);
```

Listing - destrukturiranje i spread/rest operator

U navedenom primeru varijabli `a` je dodeljena prva vrednost niza, varijabli `b` druga vrednost niza, a varijabli `c` niz kreiran od svih preostalih vrednosti, odnosno `[3,4,5]`.

U strukturiranoj dodeli vrednosti moguće je zadati default vrednosti, koje će varijable dobiti ukoliko destrukturiranje ne uspe. Lisitng ispod ilustruje ovu situaciju.

```
var [a=11,b=22,c=33] = [1,2];

console.log(a, b, c);

var {x:x1=11,z:z1=22,w:w1=44} = {x:1,y:2,z:3};
console.log(x1, z1, w1);
```

Listing - default vrednosti strukturirane dodele

U slučaju niza varijabla `c` će dobiti vrednost `33`, zbog toga što u nizu ne postoji treći element. U slučaju objekta, varijabla `w1` će dobiti vrednost `44` jer objekat koji se destrukturira nema atribut `w`.

Koncizni atributi i metode objekata

Od verzije ES6 moguć je skraćen zapis objekata i metoda. Primer je dat listingom ispod.

```
var x = 1, y = 2,

  o = {
    x,
    y,
    show(){
      console.log('x:',this.x,',y:',this.y);
    }
  }
```

Listing - koncizni atributi i metode objekta

Pogledajmo na listingu ispod kako izgleda programski kod nakon prevođenja u ES5.

```
var x = 1,

    y = 2,
    o = {
  x: x,
  y: y,
  show: function show() {
    console.log('x:', this.x, ',y:', this.y);
  }
};
```

Listing - prevedeni oblik primera sa konciznim atributima i objektima

Vidimo da koncizni zapis atributa podrazumeva da će se varijable koje se koriste u kreiranju objekta zvati isto kao i atributi za koje se postavljaju. Takođe, vidimo da u skraćenom zapisu metoda nije potrebno da pišemo atribut i ključnu reč `function`, već je dovoljno da navedemo naziv atributa/metode, listu parametara i telo funkcije. Obratite

pažnju da metoda nije kreirana kao neimenovani funkcijski izraz već je zadata definicijom funkcije. Time smo dobili mogućnost da metoda može biti rekurzivno pozvana.

Template literali (interpolirani string literali)

U ES6 moguće je zadati string literale sa embedovanim izrazima. Oni se zadaju kao višelinijski stringovi (između karaktera ’’), a izrazi se zadaju u formatu `${expression}`. Primer je dat listingom ispod.

```
function upper(s) {

      return s.toUpperCase();
}
var who = "reader"
var text =
     `Ut vitae massa ${upper( "consequat elit porttitor" )}
porttitor.
    Sed ac euismod magna. Etiam eget turpis at dolor dignissim
varius.
  Nulla hendrerit massa enim, a fringilla risus pretium nec.
  Suspendisse et ultrices dolor ${who}. `

console.log( text );
```

Listing - template literal

String `text` kreiran je tako što je `${who}` zamenjen stringom `“reader“`, a `${upper( "consequat elit porttitor" )}` zamenjen vrednošću izraza `upper( "consequat elit porttitor" )` odnosno stringom `"CONSEQUAT ELIT PORTTITOR"`.

Arrow funkcije

Arrow funkcijski izrazi su skraćeni zapis funkcijskih izraza koji nemaju sopstvene vrednosti za `this`, `arguments`, `super` i `new.target`, već koriste vrednosti iz leksičkog opsega funkcije.

Sintaksa arrow funkcije data je listingom ispod.

```
(param1, param2, …, paramN) => { statements }
```

Listing - sintaksa arrow funkcije

Arrow funkcija počinje listom parametara u zagradama. Zatim sledi strelica =>, pa telo funkcije u vitičaste zagradama. Obratite pažnju da arrow funkcija ne ostavlja mogućnost zadavanja imena - arrow funkcije su uvek neimenovani funkciji izrazi. Ukoliko telo funkcije ima samo jedan izraz, ne moramo ga obuhvatiti u vitičaste zagrade - ako izostavimo vitičaste zagrade vrednost jedinog izraza funkcije će automatski biti vraćena iz funkcije.

Pogledajmo ES5 kod u listingu ispod.

```
function Prefixer(prefix) {

    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    var that = this; // zbog cega nam treba ova linija koda?
    return arr.map(function (x) {
        return that.prefix + x;
    });
};

var pre = new Prefixer('Zdravo ');
console.log(pre.prefixArray(['Marko', 'Luka']));
```

Listing - this u ugnježdenoj funkciji

Krairali smo jednu kosntruktorsku funkciju - `Prefixer` - koja kreira objekat i postavi mu vrednost atributa `prefix`. U prototip smo dodali metodu `prefixArray` koja primi niz `arr` i funkcijom `map` na svaki string u nizu doda postavljeni prefiks. Obratite pažnju na liniju u kojoj smo postavili `var that = this;`. Zbog čega nam je to bilo potrebno? Setimo se da je `this` u ugnježdenoj funkciji globalni objekat, a ne objekat nad kojim je definisana metoda. Da bismo mogli da pristupimo objektu u kom je definisana metoda i u okviru ugnježdenih funkcija, moramo napraviti lokalnu varijablu koja će imati referencu na taj objekat. Zahvaljujući leksičkom opsegu vidljivosti varijabli, toj varijabli možemo da pristupimo iz ugnježdene funkcije.

Istu funkcionalnost mogli smo postići jednostavnije da smo koristili arrow funkciju, što je prikazano listingom ispod.

```
function PrefixerA(prefix){

    this.prefix = prefix;
}

PrefixerA.prototype.prefixArray = function (arr) {
     //this ima opseg
    return arr.map(x => this.prefix + x);
};

pre = new PrefixerA(Zdravo ');
console.log(pre.prefixArray(['Marko', 'Luka']));
```

Listing - Arrow funkcije i `this` objekat

Činjenica da `arrow` funkcija nema sopstveni `this` objekat već koristi this iz svog leksičkog opsega omogućuje nam da izbegnemo potrebu za pravljenjem lokalne varijable sa referencom na `this`.

for ... of

U ES6, pored iteriranja kroz indekse niza moguće je i iteriranje kroz vrednost niza. Za to se koristi `for ... of` konstrukcija, kao što je prikazano listingom ispod.

```
var a = ["a","b","c","d","e"];

for (var idx in a) {
  console.log( idx );
}
// 0 1 2 3 4
for (var val of a) {
  console.log( val );
}
// "a" "b" "c" "d" "e"
```

Listing - for ... of

Simboli

Specifikacija ES6 uvodi novi primitivni tip podataka - simbol - koji služi za reprezentovanje jedinstvenih identifikatora i najčešće se koristi u kombinaciji sa konstantama. Svaka vrednost vraćena pozivom funkcije Symbol() je garantovano jedinstvena što simbole čini pogodnim da budu identifikatori atributa objekta. Listing ispod daje primer takvog korišćenja simbola.

```
const MY_KEY = Symbol();

const FOO = Symbol();
const obj = {
    [MY_KEY]: 123,
    [FOO]() {
        return 'bar';
    }
};
console.log(obj[MY_KEY]);//123
console.log(obj[FOO]());//bar
```

Listing - simbol

Iteratori

Iteratori u ES6 omogućuju jednostavan prolazak kroz kolekcije. Objekat je iterator ako je zadato kako da se kreće kroz kolekciju pristupajući jednom po jednom elementu i vodeći računa o trenutnom elementu. Shodno tome iterator obavezno ima `next()` metodu koja prelazi na sledeći element u kolekciji i vraća ga. Pored ove metode, iterator može da ima i `return()` metodu koja vraća trenutni element kolekcije i završava iteriranje. Povratna vrednost iteratora je objekat koji ima dva atributa: `value`, vraćenu vrednosti i `done`, boolean indikator završenosti iteracije.

Iteratori mogu da se koriste u kombinaciji sa `for ... of` konstruktom i tada u for petlji prolazimo kroz jedan po jedan element kolekcije. Da bi to bilo omogućeno, potrebno je da objekat ispoštuje *iterable* protokol. Iterable protokol nalaže da objekat na atributu `[Symbol.iterator]` ima funkciju koja vraća iterator (objekat koji implementira `next()`). Listingom ispod dat je iterator kroz Fibonačijeve brojeve.

```
var Fib = {

  [Symbol.iterator]() {

    var n1 = 1, n2 = 1;

    return {

      next() {
        var current = n2;
        n2 = n1;
        n1 = n1 + current;
        return { value: current, done: false };
      },

      return(v) {
        console.log("Fibonacci sequence abandoned.");
        return { value: v, done: true };
      }
    };
  }
};

for (var v of Fib) {
  console.log( v );
  if (v > 50) break;
}
```

Listing - iterator

Vidimo da je Fib objekat koji `[Symbol.iterator]` ima funkciju koja vraća objekat sa dve metode: `next` računa sledeći Fibonačijev broj i `return` koja prekida iteriranje. Sada je jasno zašto niz u ES6 dozvoljava `for ... of` operator: u ES6 niz implementira iterable protokol. Obratite pažnju da Fibonačijevi brojevi nisu konačna kolekcija!

Generatori

Generatori su funkcije čije izvršavanje može da se pauzira i da se kasnije nastavi. Ciklus pauziranja/nastavljanja omogućuje dvosmernu razmenu poruka. Generator može vrati vrednost u toku svog izvršavanja ali i kod koji poziva generator može generatoru da prosledi vrednost.

Sintaksa zadavanja generatora data je listingom ispod.

```
function* gen() {

  yield 1;
  yield 2;
  yield 3;
}

var g = gen();
```

Listing - sintaksa generatora

Na listingu iznad vidimo da se generator zadaje gotovo identično kao i funkcija. Obratite pažnju na asterisk (zezdicu) koji sledi nakon ključne reči `function`. Ključna razlika u odnosu na funkcije je što se povratna generatora vrednost vraća pomoću `yield` umesto `return`, tako da može biti više povratnih vrednosti.

Poziv generatora (u našem slučaju `gen()`) vraća objekat koji implementira i iterator i iterable protokol. To znači da se nad pozvanim objektom može pozvati `next`, ali i da se može iterirati u `for ... of` petlji.

Kao i u slučaju iteratora, kolekcija kroz koju se prolazi generatorom ne mora biti konačna. Pogledajmo primer ispod u kom je generator iskorišćen za kreiranje autoinkrementirajućih identifikatora.

```
function* idMaker() {

    var i = 0;
    while(true)
        yield i++;
}

var gen = idMaker();
for(let id of gen){
  console.log(id);
  if(id==15){
    break;
  }
}
```

Lisitng – autoinkrementirajući identifikator

Napravili smo generator `idMaker` koji u svakoj iteaciji trenutnu vrednost uveća za 1. Inicijalna vrednost je 0. U for ... of petlji smo ispisali prvih 15 `id`-ova.

Moduli

Moduli u JavaScriptu su objekti koji imaju javni deo, koji je vidljiv od spolja i privatni, koji nije vidljiv od spolja. Moduli su od uvek bili deo JavaScripta, slobodno možemo reći čak da su oduvek bili osnovni šablon strukturiranja koda. Podsetimo se da smo mogli jednostavno da ih implementiramo kao funkciju koja vraća objekat. Privatna svojstva su bila lokalne promenljive funkcije, a javna svojstva bili bi atributi objekta. Međutim, ES6 donosi novi sintaktički oblik zapisivanja modula. Pri tome svaki modu treba da bude zaseban fajl. Ovako kreirani moduli imaju statički definisan API, koji se zadaje kroz eksporte i kasnije ne može da se izmeni. Važno je napomenuti da su moduli singlton objekti, odnosno da se, prilikom više učatavanja istog modula dobija referenca na jedan isti objekat.

Varijable i funkcije se u javni API dodaju ključnom rečju `export`. Sve funkcije i varijable koje nisu eksplicitno eksportovane tretiraju se kao da su privatne. Pogledajmo primer u listingu ispod[4].

```
//dummyModule.js

var foo = 42;
var baz = function () {
      return 'first value';
};
export default { foo };
export var bar = "hello world";
export {baz};

//primer10.js
// import {default as foo, bar, baz} from './dummyModule';
import foo, {bar, baz} from './dummyModule';
console.log('foo:',foo);
console.log('bar:',bar);
console.log('baz:',baz());

import * as dummy from './dummyModule';
console.log('default:',dummy.default);
console.log('baz:',baz());
```

Listing – modul

Navedeni primer zahteva objašnjenje. U fajlu `dummyModule.js` kreirana je varijabla `foo` koja ima vrednost `42` i funkcija `baz` koja vraća string vrednost `'first value'`. Zatim je eksportovana varijabla `foo`. `Default` znači da je to glavni eksport modula. Nakon toga je eksportovana nova varijabla `bar` koja je string `'hello world'`. Na kraju je eksportovana funkcija `baz`. API navedenog modula dat je listingom ispod.

```
{

  bar:"hello world"
  baz:function baz()
  default:42
}
```

Listing – API modula

Uvoz modula realizuje se ključnom rečju `import`. Obzirom da modul eksportuje objekat import se realizuje pomoću destrukturiranja. Jedino `defualt` eksport može da se uveze direktnim navođenjem imena. Takođe moguće je uvesti i ceo modul. Primer uvoza navedenog modula dat je listingom ispod.

---

[4] U vreme pisanja ovih materijala pregledači ne podržavaju `require` funkciju za import modula. Primer 10 se može pokrenuti pomoću komande `browserify primer10-uc.js -o primer10.js -t [babelify]` koja, naravno zaheva da je `browserify` npm paket instaliran.

```
import foo, {bar, baz} from './dummyModule';


import * as dummy from './dummyModule';
```

Listing - import modula

Obzirom da je `foo` default eksport modula, importovali smo ga bez potrebe za destrukturiranjem. Varijable `bar` i `baz` dobili smo strukturiranom dodelom vrednosti. Poslednja linija koda pokazuje kako je moguće importovati čitav modul.

Klase

Prototipska priroda JavaScripta sa jedne strane i konstrukti kao što su `new`, `.constructor` i `interface` sa druge upućuju na ambivalentni odnos tog jezika prema klasičnom OOP. Od ES6 ovaj odnos je još kompleksniji zahvaljujući činjenici da je uveden i sintaktički konstrukt `class`. Primer zadavanja klasa dat je listingom ispod.

```
class Foo {

  constructor(a,b) {
    this.x = a;
    this.y = b;
  }
  gimmeXY() {
    return this.x * this.y;
  }
}
var f = new Foo( 5, 15 );
```

Listing - klasa

Klasa se zadaje ključnom rečju `class` nakon čega sledi naziv klase. Konstruktor je opcioni i, ako se ne zada, automatski će se zadati default konstruktor bez parametara. Važno je da napomenemo da da se klase ne hoistuju, odnosno da definicija klase mora u kodu da prethodi korišćenju klase. Obratite pažnju na način na koji je klasa instancirana: pri tome se uvek koristi ključna reč `new`. U stvari, klasa u ES6 može da se posmatra kao *koncizan zapis konstruktorske funkcije* u ES5.

Navedena klasa ima dva javna atributa: `x` i `y`, i jednu javnu metodu: `gimmeXY` koja vraća proizvod `x` i `y`. Sve lokalne varijable konstruktora koje nisu dodate u `this` objekat tretiraju se kao privatna svojstva.

Pored definisanja klasa možemo i da imamo nasleđivanje, koje se ostvaruje pomoću ključne reči `extends`, čime se uspostavlja referenca prototipa. U klasi naslednici konstruktoru roditeljske klase možemo da pristupimo ključnom rečju `super`. U konstruktoru naslednice nije moguće pristupiti objektu `this` pre nego što se pozove `super` konstruktor roditeljske klase. Listingom ispod dat je primer nasleđivanja u ES6.

```
class Bar extends Foo {

  constructor(a,b,c) {
    super( a, b );
    this.z = c;
  }
  gimmeXYZ() {
    return super.gimmeXY() * this.z;
  }
}

var b = new Bar( 5, 15, 25 );
```

Listing – nasleđivanje

Listingom iznad dat je kod u kom smo kreirali novu klasu `Bar` koja je naslednica klase `Foo`. Ova klasa uvodi novi javni atribut `z` i novu javnu metodu `gimmeXYZ`.

Takođe, moguće je definisati i statička svojstva, korišćenjem ključne reči `static`.