Nya egenskaper i ECMAScript 2015 (ES6) och deras inverkan på JavaScript-programmering

Mathias Fredriksson, [MATRIKEL]

Kandidatavhandling i Datavetenskap

Handledare: Annamari Soini

Fakulteten för naturvetenskaper och teknik

Åbo Akademi

2016

Referat

Denna avhandling omfattar JavaScript, ECMAScript, deras historia och nya egenskaper som introducerats i den senaste upplagan av ECMAScript, ECMAScript 2015. Det ställs många krav på JavaScript idag som världens mest använda programmeringsspråk. Med avsikt på detta undersöker vi vad de nya egenskaperna är och vad de har för betydelse, både för språket och i sammanhanget av att programmera JavaScript. Syftet med denna avhandling är att läsaren skall förstå vad de nya egenskaperna är och vad de löser för problem, inte att lära läsaren syntax eller praktisk användning.

Nyckelord: JavaScript, ECMAScript, programmering, programmeringsspråk, standardspecifikation

Innehållsförteckning

1 Inledning

JavaScript uppfanns på 10 dagar och är idag världens mest använda programmeringsspråk. Det stöds av nästan alla webbläsare som existerar och är de facto programmeringsspråket som används för att utveckla interaktiva webbsidor. Idag är det snarast ett undantag att en webbsida inte är beroende av JavaScript för en stor del av dess funktionalitet. Utöver det har JavaScript sett en markant ökning i användningsområden tack vare den populära JavaScript plattformen Node.js. Node.js är en JavaScript-körtid, baserad på Chrome V8 JavaScript-motorn, som möjliggör exekvering av JavaScript-kod utanför webbläsaren. I och med Node.js har det blivit populärt, och framförallt lätt, för programmerare att utveckla både klient- (användargränssnitt) och server-programvaran i JavaScript.

Dessa nya användningsområden ställer krav på programmeringsspråket, krav som inte funnits från första början. Hur kan dessa krav åtgärdas i ett språk som är så allestädesnärvarande som JavaScript? I denna avhandling ser vi på historian av både JavaScript och ECMAScript, vad ECMAScript har för relevans då vi talar om JavaScript och planen för att åtgärda brister i programmeringsspråket. Därefter redogör vi för en del av de nya egenskaper som introducerats i den senaste upplagan och vad dessa egenskaper har för relevans. Vi avslutar med en diskussion för att sammanfatta uppdateringen till programmeringsspråket och hur lyckat det var.

2 Historia

2.1 Från JavaScript till ECMAScript

JavaScript-programmeringsspråket har sitt ursprung hos Netscape där det utvecklades av Brendan Eich i maj 1995 på endast 10 dagar. I början kallades språket Mocha men namnet byttes till LiveScript och snart därefter till JavaScript efter att Sun Microsystems gett licensen för att använda varumärket. År 1996 togs JavaScript-språket till Ecma International för att utarbeta en standardspecifikation som kunde implementeras av andra webbläsare. Arbetet hos Ecma International ledde till första upplagan av standardspecifikationen ECMA-262. Standarden fick namnet ECMAScript.

Första upplagan av ECMAScript (ES1) publicerades i juni 1997. Standarden skickades även till ISO/IEC JTC 1 och godkändes som en internationell standard under ISO/IEC 16262. Den andra upplagan (ES2) publicerades ett år senare i juni 1998 för att vara komplett med ISO/IEC 16262 standarden. Den tredje upplagan (ES3) av standarden publicerades i december 1999 och inkluderade ”reguljära uttryck, bättre stränghantering, nya kontrollpåståenden, `try`/`catch` undantagshantering, bättre definitioner på fel, formatering för numerisk utdata och små förändringar förutseende framtida växt för språket” [1]. Det var efter publikationen av ES3 som ECMAScript i kombination med webben såg ett massivt ibruktagande och blev de facto det programmeringsspråk som används på webben.

2.2 ECMAScript 4 och framåt

En stor strävan till att förbättra språket påbörjades år 2000 med ECMAScript 4 (ES4). Det var en långsam process och de intresserade parterna hade svårt att komma överens om språkets framtid. Redan år 2003 tog arbetet på ES4 slut. Ett par år senare, i februari 2005, publicerade Jesse James Garrett en vitbok om en ny metod, Ajax (asynkron JavaScript + XML). Snart därefter röstade TC39 att fortsätta arbetet på ES4, fortfarande baserat på den 7 år gamla ES3. TC39 kommitteen var trots detta tudelad. På ena sidan fanns Adobe, Mozilla, Opera och Google som ville driva vidare ES4, på andra fanns Microsoft och Yahoo som ville göra en mindre uppdatering till språket i form av ECMAScript 3.1. I juli 2008 hölls ett möte i Oslo där kommitteen äntligen hittade samsyn om språkets framtid. Det bestämdes att språket skall uppdateras med en liten inkrementell uppdatering och därefter påbörja planeringen av en ny upplaga, en upplaga där egenskaperna från ES4 skalas tillbaka. Den nya upplagan skulle heta ECMAScript 3.1 men döptes om till ECMAScript 5.

Den nya uppföljaren till ECMAScript standarden blev märkt som ECMAScript Harmony, i anda med harmonin som kommitteen kommit fram till [2].

3 ECMAScript

3.1 Om ECMAScript

ECMAScript är namnet på programmeringsspråket som beskrivs av standardspecifikationen ECMA-262. Det är skapat av Ecma International och utveckling av språket drivs av Ecma tekniska kommitté 39, bättre känt som TC39.

3.2 ECMAScript 2015 (ES6)

ECMAScript 2015 (ES 2015), tidigare känd som ECMAScript 6 (ES6), är den 6:e upplagan av ECMAScript och den senaste officiella standardspecifikationen för språket. Standarspecifikationen godkändes den 17 juni 2015 av Ecmas generalförsamling och är den största uppdateringen till ECMAScript sedan 1999 [3]. Målen för ES 2015 är bl.a. att förbättra stödet för utveckling av vidsträckta applikationer, bibliotek och att underlätta kompilering av andra språk till ECMAScript. Bland de stora förbättringarna som gjorts till språket är ”inkludering av moduler, klasser, lexikalt blockomfång, iteratorer, generatorer, löften, nedmonteringsmönster och riktiga svansanrop. Därtill har standardbiblioteket utvidgas med stöd för nya abstraktionsmetoder för data som avbildningar, mängder och räckor med binära numeriska värden samt bredare stöd för Unicode i strängar och reguljära uttryck. Inbyggda objekt kan även utvidgas genom subklasser” [1].

TC39-kommittén har haft som mål att med ES 2015 lägga en stadig grund för framtiden av språket och göra språket mer mottagligt för att möta nya behov med en snabbare iterationsprocess för nya upplagor. Den snabbare iterationsprocessen innebär att TC39 släpper ut en ny upplaga av ECMAScript varje år med avsikt att åtgärda de behov som ställs på språket. Som följd av detta döptes ECMAScript 6 om till ECMAScript 2015 för att uppmärksamma den nya processen med årliga upplagor av språket.

3.2.1 Bakåtkompatibilitet

Ett förslag om att göra den nya ECMAScript upplagan fullständigt bakåtkompatibel introducerades av David Herman på es-discuss e-postlistan den 31 december 2011 [4]. Förslaget lade grunden för det som kallas Ett JavaScript (eng. One JavaScript) idag. Ett JavaScript innebär att sträva efter att föra språket framåt utan att introducera ändringar som inte är bakåtkompatibla. Genom att endast introducera nya programmeringsgränssnitt och ny syntax till språket undviker man att gammal kod slutar fungera. Språket skall heller inte introducera nya sätt att indikera vilken version som skall användas eller ändra på beteendet hos existerande egenskaper. Som resultat av denna strävan är ES 2015

fullständigt bakåtkompatibelt frånsett de små korrigeringar och tillägg som tas upp i Annex D och Annex E av ECMA-262 [1].

Ett JavaScript är ett bra koncept som låter språket avancera utan att söndra webben i processen, men det kommer inte utan någon olägenhet. Fullständig bakåtkompatibilitet betyder att fel i språket eller egenskaper med oväntade resultat aldrig kan åtgärdas. Utöver att felen kvarstår i språket som fallgropar för programmerare kan ECMAScript specifikationen heller inte förenklas eftersom den bör beskriva i detalj hur alla, nya och gamla, egenskaper skall bete sig.

3.2.2 Användning idag

Den nya ECMAScript-specifikationen har väckt mycket intresse bland JavaScript-utvecklare redan före dess publikation. Tack vare populariteten och det stora intresset att använda dessa nya egenskaper har webbläsartillverkaren implementerat dessa egenskaper i en snabb takt. Idag är det fullständigt möjligt att använda största delen av de nya egenskaperna från ES 2015 i de senaste versioner av alla ständigt gröna (eng. evergreen) webbläsare (Google Chrome, Mozilla Firefox och Microsoft Edge). Dessa webbläsare anses vara ständigt gröna eftersom de inte är bundna till någon version utan uppdateras automatiskt till den senaste versionen. Förutom genom stöd hos webbläsare och andra JavaScript plattformar (t.ex. Node.js) kan nya egenskaper användas med hjälp av mellanlägg (eng. shim) eller källkod-till-källkod kompilering (transkompilering). Ett mellanlägg implementerar en egenskap som fattas. Transkompilering däremot kan t.ex. förvandla ES 2015 kod till ES5 kod som sedan kan köras i så gott som alla webbläsare. Bland dessa är Babel och Googles Traceur populära alternativ. Förutom transkompilering av ES 2015 till ES5 så har det även skapats nya språk ovanpå JavaScript, så kallade ”kompilera till JavaScript”-språk. Bland dessa är TypeScript ett mycket populärt alternativ, utvecklat av Microsoft. Förutom att implementera ES 2015 egenskaper så ger TypeScript stöd för annotation av typer till JavaScript språket. Då en variabel annoteras med en typ kan variabeln i fråga inte tilldelas ett värde av en annan typ. Exempelvis skulle tilldelning av ett numeriskt värde till en variabel annoterad som en sträng resultera i ett kompileringsfel (i TypeScript).

4 Nya egenskaper

4.1 Introduktion

ECMAScript 2015 är en massiv uppdatering till språket med många nya egenskaper och förbättringar. För att bättre förstå vad detta betyder, både för språket och programmerare, tar vi och beskriver dessa nya egenskaper, varför de introducerats och hur de förbättrar språket eller kan användas för att förbättra programmering med språket.

4.2 Blockomfång och variabler

Med blockomfång (eng. block scoping) menas att deklarationer inte läcker ut ur ett kodblock. Ett kodblock är en sektion av kod som är inom klammerparenteser och har i tidigare versioner av JavaScript skapats genom funktionsdeklarationen `function`. Genast anropade funktionsuttryck (eng. immediately invoked function expression, IIFE) är en konvention som använts för att skapa dessa blockomfång.

```
(function IIFE() { var message = ”Gömd variabel”; })();
```

Exemplet ovan demonstrerar hur en variabel `message` som är privat till funktionen `IIFE()` kan skapas med ett genast anropat funktionsuttryck. En annan egenskap hos variabler deklarerade med `var` är att de hissas upp till toppen av ett block (funktion). Det innebär att variabeln är deklarerad för hela blocket men har inget värde innan tilldelningen som ger det ett värde nås. Eftersom beteendet av variabeldeklarationen `var` inte kan ändras för bakåtkompatibilitetsskäl har två nya, `let` och `const`, introducerats. Med `let` och `const` hissas deklarationen inte upp och den är endast giltig inom blocket där den deklareras. Det finns en betydande skillnad mellan `let` och `const`. Med `const` skapas en s.k. konstant deklaration. En `let` variabel är fri att tilldelas nya värden när som helst medan en `const` variabel endast kan tilldelas ett värde då den deklareras. Om värdet i en konstant deklaration är ett primitivt värde kan det aldrig ändra och är det ett objekt kan objektet aldrig ersättas med ett annat objekt trots att innehållet i objektet fortsättningsvis kan modifieras.

Blockomfång sker på alla ställen där ett kodblock är inom klammerparenteser och `let` eller `const` används. Detta påverkar exempelvis `if`-satser och `for`-loopar. Fördelen med detta är att det bättre går att förmedla en programmerares avsikt med en variabel eftersom variabeln inte existerar utanför dess blockomfång. Med `const` är det ytterligare möjligt att förmedla avsikt genom att en variabel aldrig kommer att ändra. Detta förebygger också fel där en variabel av misstag blir tilldelat ett felaktigt värde.

4.3 För-av loopen

För-av loopen (eng. for-of loop), eller ”för variabel av iterator”-loopen, är en ny syntax för att iterera över element av en iterabel. Loopen skrivs som `for (let item of collection)` där `collection` är en iterabel och `item` är en variabel som får ett nytt värde från iteratorn av iterabeln i varje iteration av för-av loopen. Eftersom tidigare versioner av ECMAScript redan definierade en för-i loop (eng. for-in loop) kan det frågas varför en ny syntax behövs för iteratorer. För-i loopens syfte är att iterera över räknebara egenskaper i ett objekt utan definierad ordning. Den kan inte användas för att pålitligt iterera över en räcka eftersom ordningen inte är garanterad och tomma element i en räcka tas inte i beaktande. Om syftet är att endast iterera över objektets egna egenskaper måste egenskapen verifieras t.ex. genom `if (obj.hasOwnProperty(prop))`. Dessa egenskaper hos för-i loopen gör den opassande för att iterera över iterabler och skulle orsaka problem med bakåtkompatibilitet. Exempelvis skulle resultatet från att iterera över en räcka (som i nya standarden är en iterabel) inte vara det samma som tidigare [5] [6].

En begränsning hos för-av loopen är att det inte går att iterera över vanliga objekt eftersom vanliga objekt inte är iterabler. Det är möjligt att kringgå begränsningen genom att implementera iterabel-gränssnittet för objektet eller genom att skapa en iterator för objektet, vilken används i för-av loopen istället för objektet. I ECMAScript 2017 åtgärdas detta genom att introducera två nya egenskaper för objekt, `Object.values()` och `Object.entries()` [7]. De nya egenskaperna fungerar som komplement till den redan existerande `Object.keys()` och möjliggör att lätt iterera över nycklar, värden samt par av nyckel och värde i ett objekt.

4.4 Löften för asynkron programmering

Löften (eng. promises) har under de senaste åren blivit en väsentlig del av JavaScript-programmering. De underlättar programmerare med både flödeskontroll samt resonemang av asynkron kod. Ett löfte kan ses som en platshållare för ett framtida värde. Det enda vi vet om denna platshållare är att det vid något skede kommer att ha ett värde och att operationen endera har lyckats eller inte. Med hjälp av dessa garantier kan vi koppla beteende till löften och låtsas som om värden redan fanns tillgängliga.

För att bättre förstå löften skall vi se på vad det allmänna sättet att hantera asynkron kod har varit före dem. Tänk dig en funktion som tar emot en annan funktion som sedan kallas på då den första funktionen är färdig, vi kallar den andra funktionen en en återkallande funktion (eng. callback function). Denna stil av programmering kallas Continuation Passing

Style (CPS) och innebär att en funktion förses med en fortsättning, fortsättningen kallas sedan på av funktionen med resultatet. Fortsättningen i detta sammanhang syftar på en återkallande funktion. Principen här är att en funktion kan meddela sitt resultat genom en återkallande funktionen istället för att ge resultatet som ett returvärde. Det är lätt att förstå den grundläggande mekanismen bakom återkallande funktioner men programflödet är inte sekventiellt, vilket gör det svårt att förstå vilka delar av ett program exekveras i vilken ordning. Här är det viktigt att beakta att kontrollen ges över till funktionen i fråga, det finns varken några garantier för hur den återkallande funktionen kallas på eller om den över huvud taget kallas på. Med löften tas denna kontroll tillbaka. Löften ersätter inte återkallande funktioner i sin helhet, men löser många av de problem som förekommer med dem.

Kris Zyp gav ett förslag den 24 mars 2009 för ett programmeringsgränssnitt för löften på Google gruppen för CommonJS [8]. Förslaget blev kallat Promises/A [9] och har i sin tur varit en grund till arbetet bakom Promises/A+ standarden. Promises/A+ är en öppen standard för löften inom JavaScript som kan fungera sinsemellan [10]. Första versionen av standarden publicerades i slutet av 2012 och har därefter uppdaterats två gånger. Den senaste versionen av standarden (1.1.1) publicerades den 5 maj 2014 [11] och idag är många implementationer av löften baserade på Promises/A+ standarden [12]. Arbetet bakom Promises/A+ standarden har enat JavaScript gemenskapen bakom en standard för hur löften skall bete sig.

För göra det mesta av löften i ett språk som JavaScript med många tredjeparts-bibliotek bör det finnas en väldefinierad standard för hur de skall fungera, oberoende av implementation. Detta har åstadkommits via Promises/A+ och nu den ny ECMAScript standarden.

Exempel på hur en återkallande funktion kan uttryckas med löften:

4.5 Iterabler och iteratorer

Iteratorer är ett programmeringskoncept som introducerats år 1976 i programmeringsspråket CLU [13]. En iterator kan användas för att hämta data ur en datakälla, ett element i taget. Iteratorn kan liknas med en databaspekare, den returnerar ett resultat i taget till det inte längre finns resultat att returnera, likaså gör en iterator. En iterabel kan i sin tur returnera en iterator som returnerar data från iterabeln i fråga. Iterabler och iteratorer är två av de nya egenskaperna som introducerats i ES 2015. De kan implementeras på alla objekt i JavaScript genom att uppfylla de implicita gränssnitten som

definieras av standarden. Vi kallar gränssnitten implicita eftersom ES 2015 inte introducerat konceptet av gränssnitt och de finns inte tillgängligt i kod.

För att förstå hur väsentliga iterabler och iteratorer är för JavaScript måste vi introducera två koncept. Producenter av data och konsumenter av data. Iterabler och iteratorer hör till den första kategorin, de producerar data. Till kategorin data konsumenter har det introducerats flera nya egenskaper. Det finns en ny syntax för loopar (se Blockomfång och variabler) som kan ta emot en iterabel och automatiskt skapa dess iterator och loopa över elementen. Förutom för-av loopen så används iterabler även av spridningsoperatorn, löften, generatorer samt vid nedmontering och konstruktion av räckor, avbildningar och mängder.

4.6 Generatorer

Precis som iteratorer har generatorer också introducerats av programmeringsspråket CLU redan år 1975 [13]. En generator är en funktion som kan göra ett avbrott i sin exekveringen och fortsätta från samma ställe vid ett senare skede. Traditionellt exekveras all kod i JavaScript från topp till botten, dvs. en funktion exekveras från början till slut innan någon annan kod får en chans att exekveras. Generatorer vänder denna princip upp-och-ner och introducerar en helt ny programmeringsparadigm till JavaScript. Med en asterisk i funktionsdeklarationen (`function *generator() {}`) kan vi skapa en generator som fungerar väldigt olikt än vanlig funktion. Det går inte att direkt anropa en generator som en vanlig funktion. Den börjar inte sin exekvering direkt då den anropas, istället returnerar den en iterator som har en egen instans av funktionen. Med iteratorn är det möjligt att påbörja generatorns exekvering genom att kalla på `next()`-metoden. För att styra generatorer har två nya nyckelord, `yield` och `yield *`, introducerades. Nyckelorden används både för att göra avbrott i exekveringen och att passera data. Varje `yield` nyckelord är en chans att skicka data två vägar, både från och till generatorn. Då `yield` används returneras värdet efter nyckelordet och generatorn gör ett avbrott i sin exekvering. Generatorns iterator kan därefter använda sin `next()`-metod för att förse generatorn med ett värde. Praktiskt taget ersätts `yield` nyckelordet i generatorn med värdet från `next()`-metoden och generatorn fortsätter därefter sin exekvering.

Det finns många användningsområden för generatorer. Exempelvis kan de användas för att generera data efter behov. Då behöver inte minnet allokeras med alla framtida värden vilket i sin tur ger oss möjligheten att generera data i oändlighet. Generatorer kan även användas för att skriva asynkron kod som verkar som synkron för läsare. Om vi tänker oss

en generator som `yield`:ar ett löfte och gör ett avbrott. Vi kan då utanför generatorn, då löftet uppfylls, ge resultatet tillbaka till generatorn genom iteratorns `next()`-metod. I detta exempel verkar koden synkron för läsaren, resultatet finns tillgängligt i generatorn direkt efter `yield`. Ett populärt sätt att göra det är med s.k. körare (eng. runners) som automatiserar passandet av löften och fortsätter exekveringen av generatorn.

4.7 Nedmontering

[WIP] Nedmontering (eng. destructuring) är en ny egenskap som gör det lättare att definiera variabler genom att extrahera data från objekt, räckor och iterabler.

4.8 Spridningsoperatorn

Spridningsoperatorn (eng. spread operator), betecknat av tre punkter (`...`), är en ny operator som utnyttjar iterabler och iteratorer för att sprida ut en samling (t.ex. en räcka) på plats. Om vi har en räcka på 3 element, `[1, 2, 3]`, och sprider ut den då vi anropar en funktion, `bearbeta(...[1, 2, 3])`, resulterar det i att `bearbeta` anropas med tre parametrar istället för en (räckan). Det egentliga funktionsanropet set ur som `bearbeta(1, 2, 3)`. Detta är inte endast begränsat till funktionsanrop, utan kan även användas då vi skapar nya samlingar eller tilldelar värden till variabler.

4.9 Restoperatorn

Restoperatorn (eng. rest operator), betecknat av tre punkter (`...`), kan användas både i funktioner och vid nedmontering för att sampla ihop resterande parametrar under en och samma variabel. Variabeln som bildas av av operatorn blir en räcka. Detta är både en behändig operator för programmerare och löser samtidigt ett riktigt problem i tidigare upplagor av språket. För att hantera en okänd mängd parametrar till en funktion har `arguments`-variabeln varit det ända sättet. Variabeln är ett objekt som liknar en räcka och är tillgänglig lokalt i varje funktion. Problemet som restoperatorn löser är att vi får en riktig räcka och en explicit variabeldeklaration istället för den implicita `arguments`-variabeln. Med restoperatorn behöver vi heller inte filtrera bort tidigare parametrar från `arguments` om vi redan definierat dem som denna funktions deklaration `function(value, ...args) {}` demonstrerar. Här definierade vi variabeln `value` och den kommer inte att var en del av variabeln `args`.

4.10 Förvalda värden

[WIP] Förvalda värden (eng. defaults) kan användas både i funktioner och vid definition av variabler. Dessa värden låter en programmerare på förhand bestämma vilket värde en parameter skall ha om den inte definieras ett annat värde.

4.11 Klasser

[WIP] En ny egenskap i den senaste upplagan är klasser (eng. class) men till skillnad från klasser bekanta från andra objektorienterade programmeringsspråk så är ECMAScript klasser endast ett syntaktiskt socker. I bakgrunden omvandlas dessa klasser till objekt.

4.12 Pilfunktionen

Pilfunktionen (eng. arrow function) har introducerat en ny syntax för att skapa funktioner med egenskaper som skiljer sig från en vanlig funktion. Till dessa egenskaper hör kortare syntax gentemot en vanlig funktion, lexikalt bindande av `this` och de är alltid namnlösa (anonyma funktioner). Den kortare syntaxen för pilfunktionen kan vara lockande att använda i syftet att sparar tecken men skall inte misstas som en ersättare för ett vanligt funktionsuttryck. Exempelvis kan pilfunktionen inte fungera som en generator. I ECMAScript refererar `this` till omgivningen (objektet) en funktion körs i och kan liknas med en klassfunktion i Java som refererar till `this`. Till skillnad från Java kan en funktion i ECMAScript ha många olika `this` beroende på hur funktionen blir kallad på. Med ett lexikalt bindande av `this` menar vi att `this` inte längre refererar till omgivningen utan kontexten där den definieras. I tidigare versioner av språket används ofta konventionen `var self = this;` för att kringgå ett icke-lexikaliskt `this`. Konventionen tillåter att variabeln `self` används istället för `this` och på så sätt referera till den korrekta omgivningen. Alternativt kan en funktion också bindas till ett specifikt `this`. Pilfunktionen löser detta problem genom att inte definiera en egen `this` och tillåter heller inte att `this` binds om. Som vi ser gör pilfunktionen det lättare att resonera kring vad `this` betyder i ett specifikt kontext men kan i sin tur ha introducerarat nya fallgropar för programmerare. Pilfunktionens syntax påminner mycket om lambdafunktioner som ofta förekommer i funktionella språk och har sina rötter i det formella systemet λ-kalkyl (lambdakalkyl) som utvecklades av Alonzo Church under 1930-talet [14]. Den korta syntaxen för pilfunktioner gör dem ypperliga att användas i simpla funktioner som tar emot ett argument. Följande exempel demonstrerar fyra olika sätt att skriva pilfunktioner:

```
1. x => x * x
2. (x, y) => x * y
```

```
3.  (x, y) => { x = modifiera(x); return x * y; }
4.  x => ({ x: x })
```

Exempel 1 är det kortaste sättet och består av bara ett argument och ett returvärde. I detta fall är returvärdet implicit och då behövs inte nyckelordet `return`. I exempel 2 har vi två parametrar och då krävs det parenteser runt dem. Den mera avancerade funktionskroppen i exempel 3 kräver ett block och då är inte returvärdet implicit längre. Om nyckelordet `return` lämnas bort i exempel 3 kunde det lätt gå obemärkt eftersom funktionen är fullständig men returnerar inget värde. Exempel 4 returnerar ett objekt och då måste objektet omringas av parenteser. Om parenteserna togs bort från exempel 4 skulle objektet tolkas som en funktionskropp och vara en fullständig funktion som inte returnerar ett värde. Alla dessa regler för att skriva pilfunktionsuttryck kan ses som en kostnad för den korta syntaxen.

4.13 Avbildningar och mängder

[WIP] Avbildningar (eng. map) och mängder (eng. set) är två nya datastrukturer i den nya ECMAScript standarden.

4.13.1 Svaga avbildningar och svaga mängder

Svaga avbildningar (eng. WeakMap) och svaga mängder (eng. WeakSet) är i stort sätt likadana dom de icke-svaga varianterna. Skillnaden är att svaga varianterna håller sin referens svagt. Detta betyder att ett objekt kan frias från minnet även om det refereras till av en svag avbildning eller en svag mängd så länge som ingen annan del av programmet refererar till objektet.

4.14 Symboler

Symbolen (eng. symbol) är det första nya primitiva typen i ECMAScript sedan språket standardiserades. Symboler är värden, precis som strängar, nummer och objekt är värden. Däremot har symboler unika egenskaper som inte kan liknas med andra typer. Värdet för en symbol är oföränderligt och garanterar att det inte finns ett annat värde som är likadant. Syftet med symboler är att använda dem som egenskapsnycklar på objekt utan risk för kollision med andra nycklar. Nycklar som skapats med en symbol tas inte heller i beaktande i den gamla för-i loopen (se för-av loopen). Om vi tänker tillbaka på ECMAScript språkets bakåtkompatibilitetskrav är det lätt att förstå varför den nya typen introducerats. Symboler ger möjligheten att definiera egenskaper på objekt som varken kolliderar med andra egenskaper eller har oväntade resultat i äldre program. Deras roll i den nya standarden är

väsentlig, exempelvis skapas iterabler genom att definiera en ny egenskapsnyckel med globala symbolen `Symbol.iterator` på ett objekt.

4.15 Moduler

[WIP] Moduler (eng. modules) är inte ett nytt koncept i JavaScript. Med moduler kan

Moduler (eng. modules), eller modularisering av kod, är ett viktigt verktyg för programmerare. Det låter programmerare bryta upp helheter i mindre, separata, delar som är lätta att byta ut eller återanvända. Moduler har ofta ett utåtriktat programmeringsgränssnitt som kan användas av annan kod. Inuti modulen finns en implementation som uppfyller gränssnittet. Med moduler kan komplexa delar av program gömmas undan och så minska den kognitiva belastningen av att förstå olika delar av program.

Populära metoder för att hantera moduler är AMD (Asynchronous Module Definition) och CommonJS. Med ES 2015 finns de nu ett ytterligare sätt, standardiserat i språket.

Fördelar: Ingen mera UMD (Universal Module Definition) som är en definition för hur populära modulladdare kan funktionera sinsemellan (AMD, CommonJS). SystemJS

Standarden för laddning av moduler finns inte i ES2015, det utvecklas separat från ECMAScript standarden av WHATWG arbetsgruppen [15].

4.16 Metaprogrammering

Metaprogrammering är inte ett nytt koncept i JavaScript. Det är en form av programmering där ett program kan köra andra program eller analysera sin egen struktur.

[WIP] Den nya standarden introducerar två kraftiga verktyg för metaprogrammering, ställföreträdare (eng. proxy) och reflektionsgränssnitt(eng. reflection API).

Vidareutveckling av hur metaprogrammering kan tillämpas är utanför begränsningen av denna avhandling.

4.17 Svansanropselimination

[WIP] Svansanropselimination (eng. tail call optimization) är en optimering i rekursiv programmering där svansanrop kan utföras utan att allokera mera plats på anropsstacken [16].

4.18 Mall strängar

[WIP] Mall strängar (eng. template strings).

5 Diskussion

Vi har redogjort för den nya utvecklingsprocessen av ECMAScript standarden med kodnamnet Harmony och analyserat den senaste upplagan, ECMAScript 2015. TC39 hade som mål att förbättra språket med funktionalitet som nyttjar dagens utvecklare utan att introducera bakåtkompatibilitetsproblem. Vi såg att ECMAScript 2015 blev en väldigt stor uppdatering till språket, större än väntat, men trots detta lyckades processen och vi har äntligen en ny standard med nya egenskaper. Beaktar vi årets upplaga, ECMAScript 2016, som ennu inte publicerats ser vi att processen för att göra små, inkrementella uppdateringar till språket verkar fungera.

I denna avhandling har vi inte gjort en komplett analys av alla nya egenskaper i ECMAScript 2015, men redan från de egenskaper vi analyserat kan vi konstatera att ES 2015 är en markant förbättring över ES5 (5.1). Det är nu lättare att skriva kod med mindre oväntade resultat. Eftersom löften nu är en del av språket är det lätt att utnyttja dem utan externa bibliotek och främjar användning av dem för hantering av asynkron kod. Standardiserade moduler underlättar delning av kod och avlägsnar frågan om vilken typ av modul som bör användas. Iteratorer och generatorer har öppnat möjligheten för nya intressanta programmeringsmetoder. Det är nu lättare att handskas med data tack vare de nya datastrukturerna. Utöver detta är det lättare att både skriva och läsa kod med hjälp av pilfunktioner, nedmontering, spridning, rester och förvalda värden.

Kompabilitet för en stor del av de nya egenskaperna finns endast i de senaste webbläsare och plattformar, men med kompilering av ES 2015 till ES5 är det fullständigt möjligt att utnyttja de nya egenskaperna redan idag.

Litteraturförteckning

1. Ecma International: Standard ECMA-262.

2. Eich, B.: ECMAScript Harmony. Hämtad från es-discuss: https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html den 10 april 2016

3. Ecma International: At the June 17, 2015 Ecma General Assembly in Montreux, ECMA-262 6th edition - ECMAScript® 2015 Language Specification and ECMA-402 2nd edition - ECMAScript® 2015 Internationalization API have been adopted. Hämtad från Ecma International: http://www.ecma-international.org/news/Publication%20of%20ECMA-262%206th%20edition.htm den 3 april 2016

4. Herman, D.: ES6 doesn't need opt-in. Hämtad från es-discuss: https://mail.mozilla.org/pipermail/es-discuss/2011-December/019112.html den 3 april 2016

5. Mozilla: for…of. Hämtad från Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for.of den 5 april 2016

6. Mozilla: for…in. Hämtad från Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for.in den 5 april 2016

7. Harband, J.: ECMAScript Proposal, specs, and reference implementation Object.values/Object.entries. Hämtad från tc39 GitHub: https://github.com/tc39/proposal-object-values-entries/tree/1be53c9bab75607e24159c6921d666cd9cdbdb43 den 5 april 2016

8. Zyp, K.: CommonJS: Promise API Proposal. Hämtad från Google Groups: CommonJS: https://groups.google.com/forum/#!topic/commonjs/6T9z75fohDk den 4 april 2016

9. CommonJS: Promises/A. Hämtad från CommonJS Wiki: http://wiki.commonjs.org/wiki/Promises/A den 4 april 2016

10. Promises/A+: Specification. Hämtad från Promises/A+: https://promisesaplus.com den 4 april 2016

11. Promises/A+: Promises/A+ Changelog. Hämtad från Promises/A+: https://promisesaplus.com/changelog den 4 april 2016

12. Promises/A+: Conformant Implementations. Hämtad från Promises/A+: https://promisesaplus.com/implementations den 4 april 2016

13. Liskov, B.: A History of CLU.

14. Orendorff, J.: ES6 In Depth: Arrow functions. Hämtad från Mozilla Hacks: https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/ den 6 april 2016

15. The Web Hypertext Application Technology Working Group: Loader. Hämtad från WHATWG Github: https://whatwg.github.io/loader/ den 9 April 2016

16. Rauschmayer, D.: Exploring ES6: Upgrade to the next version of JavaScript. Leanpub