

KYLE SIMPSON

i ZAKRESY DOMKNIĘCIA

TAJNIKI JĘZYKA

JavaScript
JS

Tytuł oryginału: You Don't Know JS: Scope & Closures

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-2177-9

© 2016 Helion S.A.

Authorized Polish translation of the English edition You Don't Know JS: Scope & Closures
ISBN 9781449335588 © 2014 Getify Solutions, Inc.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/tjszak>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	5
Wprowadzenie	7
1. Czym jest zakres?	13
Teoria kompilatora	13
Poznajemy zakres	16
Zakres zagnieżdżony	21
Błędy	24
Podsumowanie	25
2. Zakres leksykalny	27
Czas lexingu	27
Oszukanie zakresu leksykalnego	30
Podsumowanie	36
3. Zakres funkcji kontra zakres bloku	39
Zakres na podstawie funkcji	39
Ukrycie w zwykłym zakresie	40
Funkcja jako zakres	44
Blok jako zakres	49
Podsumowanie	57

4. Hoisting	59
Jajko czy kura?	59
Powrót kompilatora	60
Najpierw funkcje	63
Podsumowanie	65
5. Zakres domknięcia	67
Wyjaśnienie	67
Sedno sprawy	68
Teraz mogę zobaczyć	72
Pętle i domknięcia	74
Moduły	77
Podsumowanie	85
A Zakres dynamiczny	87
B Skrypty typu polyfill dla zakresu bloku	91
Traceur	92
Niejawne kontra wyraźne bloki	93
Wydażność	95
C Leksykalne this	97
D Podziękowania	101
Skorowidz	105

Hoisting

W tym momencie powinieneś już dość dobrze znać ideę zakresu i wiedzieć, jak zmienne są dołączane na różnych poziomach zakresu w zależności od sposobu ich zadeklarowania. W przypadku zakresu zarówno funkcji, jak i bloku stosuje się dokładnie te same reguły — każda zmienna zadeklarowana wewnątrz zakresu zostaje do niego dołączona.

Jednak istnieje drobny szczegół dotyczący działania mechanizmu dołączania do zakresu, gdy deklaracje pojawiają się w różnych miejscach zakresu. I tym właśnie szczegółem zajmiemy się w rozdziale.

Jajko czy kura?

Można by uznać, że cały kod w programie JavaScript jest interpretowany wiersz po wierszu, od początku do końca w trakcie wykonywania programu. Wprawdzie pod pewnymi względami to prawda, ale pewna część przedstawionego założenia może doprowadzić do nieprawidłowych wniosków dotyczących programu.

Spójrz na poniższy fragment kodu:

```
a = 2;

var a;

console.log( a );
```

Jakiego wyniku oczekujesz po wykonaniu polecenia `console.log(..)`?

Wielu programistów oczekuje wartości `undefined`, ponieważ polecenie `var a` znajduje się po operacji przypisania `a = 2`; naturalne wydaje się założenie,

że zmienna `a` będzie ponownie zdefiniowana i tym samym otrzyma wartość domyślną `undefined`. Jednak dane wyjściowe ostatniego polecenia w omawianym fragmencie kodu to 2.

Spójrz na kolejny fragment kodu:

```
console.log( a );  
  
var a = 2;
```

Być może sądzisz, że skoro w poprzednim fragmencie kodu mieliśmy do czynienia z wyszukiwaniem nieco innym niż od początku do końca kodu, to wynikiem działania powyższego również będzie wyświetlenie wartości 2. Część programistów może pomyśleć jeszcze inaczej: ponieważ zmienna `a` została użyta wcześniej, nowy fragment kodu spowoduje zgłoszenie błędu typu `ReferenceError`.

Niestety, oba założenia są nieprawidłowe. Wynikiem działania powyższego fragmentu kodu jest wyświetlenie wartości `undefined`.

Powstaje pytanie: co się tutaj dzieje? Wygląda na to, że mamy tu odwieczny problem „jajko czy kura”. Co było pierwsze: deklaracja („jajko”) czy przypisanie („kura”)?

Powrót kompilatora

Aby odpowiedzieć na powyższe pytanie, musimy powrócić do rozdziału 1. i przedstawionej tam analizy dotyczącej kompilatorów. Przypomnij sobie, że silnik w rzeczywistości przeprowadza kompilację kodu JavaScript przed jego interpretacją. Jedną z faz etapu kompilacji jest wyszukanie i powiązanie wszystkich deklaracji z odpowiednimi zakresami. W rozdziale 2. dowiedziałeś się, że stanowi to serce zakresu leksykalnego.

Dlatego też najlepsze podejście polega na przyjęciu założenia, że wszelkie deklaracje zarówno zmiennych, jak i funkcji są przetwarzane jako pierwsze, zanim nastąpi wykonanie jakiegokolwiek innego fragmentu kodu.

Kiedy widzisz polecenie `var a = 2;`, prawdopodobnie uważasz je za pojedyncze polecenie. Jednak JavaScript traktuje je jako dwa polecenia: `var a;` i `a = 2;`. Pierwsze z tych poleceń, deklaracja, jest przetwarzane w trakcie fazy kompilacji. Z kolei drugie, przypisanie, jest pozostawione *na miejscu* i czeka na fazę wykonania.

Dlatego też pierwszy fragment kodu będzie potraktowany tak, jakby przedstawiał się następująco:

```
var a;  
a = 2;  
  
console.log( a );
```

Część pierwsza będzie wykonana w trakcie kompilacji, natomiast część druga w fazie wykonywania programu.

Z kolei drugi z omawianych fragmentów kodu będzie potraktowany, jakby przedstawiał się następująco:

```
var a;  
  
console.log( a );  
  
a = 2;
```

Dlatego też jedną z możliwości związaną z omawianym procesem, taką nieco w przenośni, jest założenie, że deklaracje zmiennych i funkcji są „przenoszone” z miejsca ich występowania na początek kodu. W ten sposób docieramy do określenia *hoisting*.

Innymi słowy, *jajko (deklaracja) było wcześniej niż kura (przypisanie)*.



Przeniesienie na początek kodu dotyczy jedynie deklaracji, natomiast wszelka logika odpowiedzialna za właściwą operację przypisania lub wykonania innego kodu *jest pozostawiona na dotychczasowym miejscu*. Jeżeli hoisting mógłby zmieniać kolejność wykonywanej logiki kodu, to mogłoby to mieć trudne do przewidzenia skutki dla programu.

```
foo();  
  
function foo() {  
    console.log( a ); // Wartość undefined  
  
    var a = 2;  
}
```

Deklaracja funkcji `foo(..)` (w omawianym przykładzie zawiera wartość rzeczywistej funkcji) zostaje poddana hoistingowi, aby jej wywołanie w wierszu pierwszym mogło być wykonane.

Trzeba koniecznie wspomnieć, że hoisting odbywa się dla *poszczególnych zakresów*. Wcześniejsze przykłady były uproszczone i uwzględniały jedynie zakres globalny, natomiast w przypadku omawianej teraz funkcji `foo(..)` mamy do czynienia z przeniesieniem deklaracji `var a` na początek funkcji (oczywiście nie na początek samego programu). Dlatego też program znacznie poprawniej można przedstawić w następujący sposób:

```
function foo() {  
    var a;  
  
    console.log( a ); // Wartość undefined  
  
    a = 2;  
}  
  
foo();
```

Jak mogłeś zobaczyć, deklaracje funkcji również podlegają hoistingowi. To jednak nie dotyczy wyrażen funkcji.

```
foo(); // Błąd typu TypeError, a nie ReferenceError!  
  
var foo = function bar() {  
    // ...  
};
```

Identyfikator zmiennej `foo` zostaje przeniesiony i dołączony do zakresu nadrzędnego (globalnego) programu. Dlatego też próba wykonania funkcji `foo()` nie kończy się błędem typu `ReferenceError`. Jednak `foo` nie ma jeszcze przypisanej wartości (byłaby, gdyby zamiast wyrażenia funkcji została użyta deklaracja zwykłej funkcji). Próba wywołania `foo()` kończy się otrzymaniem wartości `undefined`, co oznacza niedozwoloną operację i zgłoszenie błędu typu `TypeError`.

Przypomnij sobie, że nawet w przypadku nazwanego wyrażenia funkcji nazwa identyfikatora pozostaje niedostępna w zakresie nadrzędnym:

```
foo(); // Błąd TypeError  
bar(); // Błąd ReferenceError  
  
var foo = function bar() {  
    // ...  
};
```

Powyższy fragment kodu będzie (wraz z hoistingiem) zinterpretowany w następujący sposób:

```
var foo;

foo(); // Błąd TypeError
bar(); // Błąd ReferenceError

foo = function() {
  var bar = ...self...
  // ...
}
```

Najpierw funkcje

Hoistingowi podlegają deklaracje zarówno zmiennych, jak i funkcji. Jednak subtelny szczegół (który *może* ujawnić się w kodzie wraz z wieloma „powielonymi” deklaracjami) to fakt, że hoistingowi najpierw podlegają funkcje, a dopiero później zmienne.

Spójrz na poniższy fragment kodu:

```
foo(); // 1

var foo;

function foo() {
  console.log( 1 );
}

foo = function() {
  console.log( 2 );
};
```

Wyświetlona zostanie wartość 1 zamiast 2! Przedstawiony powyżej fragment jest przez *silnik* interpretowany w następujący sposób:

```
function foo() {
  console.log( 1 );
}

foo(); // 1

foo = function() {
  console.log( 2 );
};
```

Zwróć uwagę na powieloną (i tym samym zignorowaną) deklarację `var foo`, nawet pomimo faktu jej występowania przed deklaracją `function foo()`..., ponieważ deklaracje funkcji podlegają hoistingowi przed zwykłymi zmiennymi.

Podczas gdy wiele powielających się deklaracji `var` jest po prostu ignorowanych, każda kolejna deklaracja funkcji *nadpisuje* poprzednią.

```
foo(); // 3

function foo() {
  console.log( 1 );
}

var foo = function() {
  console.log( 2 );
};

function foo() {
  console.log( 3 );
}
```

Wprawdzie to wszystko może wydawać się jedynie interesującym rozważaniem akademickim, jednak wyraźnie podkreśla fakt, że powielone definicje w tym samym zakresie naprawdę są kiepskim rozwiązaniem i bardzo często prowadzą do dezorientujących wyników.

Deklaracje funkcji pojawiające się wewnątrz zwykłych bloków są zwykle przenoszone do zakresu nadrzędnego, a nie traktowane warunkowo, jak mogłoby wynikać z poniższego fragmentu kodu:

```
foo(); // "b"

var a = true;
if (a) {
  function foo() { console.log("a"); }
}
else {
  function foo() { console.log("b"); }
}
```

Jednak trzeba pamiętać, że to zachowanie nie należy do niezawodnych i może ulec zmianie w kolejnych wydaniach języka JavaScript. Dlatego też prawdopodobnie najlepszym rozwiązaniem jest unikanie deklarowania funkcji w blokach.

Podsumowanie

Kuszące może być potraktowanie polecenia `var a = 2`; jako pojedynczego, ale silnik JavaScript nie postrzega go w taki sposób. Zamiast tego widzi dwa oddzielne polecenia, `var a` i `a = 2`. Pierwsze jest wykonywane w trakcie fazy kompilacji, natomiast drugie w fazie właściwego wykonywania programu.

Prowadzi to do następującego wniosku: wszystkie deklaracje w zakresie, niezależnie od miejsca ich występowania, są przetwarzane *przed* rozpoczęciem wykonywania kodu. Można powiedzieć, że deklaracje (zmiennych i funkcji) są „przenoszone” na początek ich zakresów, a ten proces jest określany mianem *hoistingu*.

Hoistingowi podlegają same deklaracje, natomiast operacje przypisania, nawet przypisania wyrażeń funkcji — już *nie*.

Zachowaj ostrożność i unikaj powielonych deklaracji, zwłaszcza ich mieszania między zwykłymi deklaracjami `var` i deklaracjami funkcji — w przeciwnym razie może to być niebezpieczne!

A

abstract syntax tree, 14
anonimowe wyrażenie funkcji, 46
AST, 14

B

blok jako zakres, 49
bloki
 niejawne, 93
 wyraźne, 93
błąd ReferenceError, 40
błędy, 24

C

celowa różnica w działaniu, 100

D

domknięcie, 67, 69, 74, 79

F

funkcja, 63
 bar(), 69
 CoolModule(), 79
 eval(), 31, 35

jako zakres, 44
setTimeout(), 72
strzałki, arrow function, 97

funkcje
 anonimowe, 46
 nazwane, 46
 strzałek, 100
 typu IIFE, 73

G

globalne przestrzenie nazw, 43
gruba strzałka, 97

H

hoisting, 59, 61

I

IIFE, 47, 73
iteracja, 77
 pętli, 55

J

język Kompilatora, 17

K

- kolizje, 42
- kompilator, 13, 60
- konstrukcja
 - const, 57
 - try-catch, 51
 - with, 33, 51

L

- leksykalne this, 97
- lexing, 14, 27
- LHS, 18

M

- mechanizm
 - this, 97
 - usuwania nieużytków, 54
- metafora, 22
- moduły, 44, 77
 - nowoczesne, 81
 - przyszłe, 83

N

- nieużytki, 54

O

- operacja wyszukiwania, 29
- optymalizacja, 36
- oszukiwanie zakresu leksykalnego, 30

P

- pętla, 74
 - for, 55
 - let, 55
- projekt Traceur, 92
- przestrzenie nazw, 43

R

- RHS, 18

S

- silnik, 20
 - JavaScript, 25
- skrypty typu polyfill, 91
- słowo kluczowe
 - function, 97
 - let, 52
- specyfikacja
 - ES3, 58
 - ES6, 58
- struktura try-catch, 58

T

- teoria kompilatora, 13
- this, 97
- token, 14
- tokenizacja, 14, 27

U

- ukrycie w zwykłym zakresie, 40
- unikanie kolizji, 42
- usuwanie nieużytków, 54
- użycie
 - domknięcia, 72
 - this, 98

W

- wiązanie this, 99
- wydajność, 35, 95
- wyszukiwanie, 29
- wywoływane wyrażenia funkcji, 47
- wzorzec IIFE, 73

Z

zagnieżdżone zakresy, 28

zakres, 13, 16, 20

bloku, 39, 77, 91

domknięcia, 67

dynamiczny, 87

funkcji, 39

leksykalny, 27

na podstawie funkcji, 39

zagnieżdżony, 21

zarządzanie modułami, 44

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

DOWIEDZ SIĘ, W JAKI SPOSÓB DZIAŁA TWÓJ KOD JS!

Początkujący programista może zacząć szybko tworzyć proste aplikacje w JavaScriptcie i nabrać złudnego przekonania o swoich umiejętnościach. Jednak ten, kto chce osiągnąć prawdziwą biegłość w tym języku, musi opanować trudniejsze zagadnienia, na przykład asynchroniczność czy techniki związane z wydajnością. Choć nie jest to ani proste, ani oczywiste, trzeba też zrozumieć wewnętrzne mechanizmy rządzące JS. Dopiero wiedza na tym poziomie pozwoli na zdobycie umiejętności profesjonalisty.

Niniejsza książka to druga część serii w całości poświęconej językowi JavaScript. Jest przeznaczona dla osób, które używają JS w swojej pracy, ale postanowiły włożyć trochę wysiłku w to, aby bardzo dokładnie rozumieć, *dlaczego* i *w jaki sposób* działa ten język. Zostały tu omówione bardzo istotne koncepcje JS: zakresy i domknięcia. Poza ogólnymi informacjami szczegółowo opisano między innymi zakresy leksykalne, zakresy funkcji i bloku, mechanizm hostingu i zakresy domknięcia. Co najważniejsze, materiał jest przedstawiony w sposób przystępny, zwięzły i klarowny, ale nieodmiennie na bardzo wysokim poziomie.

Dzięki tej książce:

- poznasz najważniejsze zasady rządzące wewnętrznym sposobem działania kodu JS
- zrozumiesz pojęcie zakresu — zbioru reguł kierujących pracą silnika JavaScript
- dokładnie poznasz zagnieżdżone zakresy, czyli serie kontenerów przechowujących zmienne i funkcje
- poznasz zakresy funkcji i bloku, mechanizm hostingu, a także wzorce i korzyści płynące z ukrywania na poziomie zakresu
- zaczniesz prawidłowo stosować domknięcia w zadaniach synchronicznych i asynchronicznych, między innymi podczas tworzenia bibliotek JavaScriptu
- zbliżysz się do celu, jakim jest prawdziwe i dogłębne zrozumienie tego języka

KYLE SIMPSON

— jest Teksańczykiem, propagatorem Open Web i wielkim pasjonatem wszystkiego, co związane z językiem JavaScript. Ma dar przekazywania wiedzy, a przy tym zaraża entuzjazmem. Píše książki, prowadzi warsztaty, występuje na konferencjach o tematyce technicznej oraz pozostaje aktywnym członkiem społeczności OSS.



41877 numer katalogowy
księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>

Helion SA
ul. Kopcińskiego 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

ISBN 978-83-283-2177-9



9 788328 321779

cena: 29,90 zł

Informatyka w najlepszym wydaniu

O'REILLY®