O'REILLY®

Helion

KYLE SIMPSON

TYPY *i* SKŁADNIA

TAJNIKI JĘZYKA JavaScript

JS

Tytuł oryginału: You Don't Know JS: Types & Grammar

Tłumaczenie: Piotr Pilch

ISBN: 978-83-283-2174-8

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/tjtypy
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Typy

Większość projektantów stwierdziłaby, że język dynamiczny (np. JavaScript) pozbawiony jest *typów*. Sprawdźmy, co na ten temat napisano w specyfikacji języka ECMAScript 5.1 (*http://www.ecma-international.org/ecma-262/5.1/*):

> Algorytmy ujęte w tej specyfikacji modyfikują wartości, z których każda ma powiązany typ. Możliwe typy wartości są dokładnie tymi, które zdefiniowano w niniejszej klauzuli. Typy są dodatkowo klasyfikowane jako typy języka ECMAScript i typy specyfikacji.
>
> Typ języka ECMAScript odpowiada wartościom, które są bezpośrednio modyfikowane przez programistę używającego tego języka. Do tej kategorii typów zaliczają się typy `undefined`, `null`, `boolean`, `string`, `number` i `object`.

Jeśli jesteś zwolennikiem języków z silną (statyczną) typizacją, możesz nie zgodzić się z takim użyciem słowa „typ". W takich językach termin „typ" oznacza o wiele *więcej* niż w języku JavaScript.

Niektórzy twierdzą, że w obrębie języka JavaScript nie powinno się posługiwać terminem „typy" i że zamiast niego powinno być używane określenie „znaczniki" lub być może „podtypy".

Też coś! Użyjemy następującej ogólnej definicji (jak można sądzić, tej samej, na której bazuje sformułowanie zawarte w specyfikacji): *typ* to nieodłączny, wbudowany zbiór cech, które w unikalny sposób identyfikują zachowanie określonej wartości i odróżniają ją od innych wartości, zarówno z punktu widzenia silnika, *jak i* projektanta.

Inaczej mówiąc, jeśli silnik i projektant traktują wartość `42` (liczba) inaczej niż wartość `"42"` (łańcuch), to te dwie wartości mają inne *typy*, czyli odpowiednio typy `number` i `string`. Używając wartości `42`, *zamierzasz* wykonać jakieś działanie liczbowe (np. obliczenie matematyczne). Gdy jednak zastosujesz wartość `"42"`, *zamierzasz* zrealizować jakieś działanie łańcuchowe (np. wyświetlenie wyniku na stronie itp.). Te dwie wartości mają różne typy.

W żadnej mierze nie jest to idealna definicja. Wystarczająca jest jednak na potrzeby niniejszego omówienia, a ponadto jest spójna ze sposobem opisu typów w samym języku JavaScript.

Typ identyfikowany przez dowolną inną nazwę…

Pomińmy rozbieżności w definicji i zastanówmy się, dlaczego to, czy język JavaScript zawiera *typy*, czy nie, ma znaczenie.

Właściwe interpretowanie każdego *typu* i jego zachowania jest absolutnie kluczowe, jeśli chcemy wiedzieć, w jaki sposób poprawnie i dokładnie przekształcać wartości w wartości innych typów (zajrzyj do rozdziału 4.). Niemal każdy powstały dotychczas program JavaScript wymaga jakiejś formy obsługi konwersji wartości, dlatego ważne jest, aby przeprowadzać to w odpowiedzialny i pewny sposób.

Jeśli istnieje wartość `42` typu `number`, ale wymagasz potraktowania jej jak wartości typu `string` (np. w przypadku pobierania wartości `"2"` jako znaku na pozycji `1`), musisz oczywiście dokonać najpierw konwersji typu wartości z typu `number` na typ `string`.

Wydaje się to zupełnie proste.

Istnieje jednak wiele różnych sposobów przeprowadzenia takiej konwersji. Niektóre z nich są jawne, proste i pewne. Jeśli jednak nie zachowasz ostrożności, konwersja może przebiegać w bardzo dziwny i zaskakujący sposób.

Niejasności związane z konwersją typów to być może jeden z najpoważniejszych problemów, z jakimi zmagają się projektanci używający języka JavaScript. Często była ona krytykowana jako *niebezpieczna* i mogąca być traktowana jako błąd w projekcie języka, a tym samym kwalifikująca się do odrzucenia i unikania.

Wiedząc, czym są typy języka JavaScript, zamierzamy pokazać, dlaczego *zła reputacja* konwersji typów jest w dużej mierze wynikiem przesadnego szumu. Dzięki temu zyskasz odpowiednią perspektywę, która umożliwi Ci dostrzeżenie możliwości i przydatności konwersji. Wcześniej jednak musimy znacznie dokładniej przybliżyć wartości i typy.

Typy wbudowane

W języku JavaScript zdefiniowano następujących siedem typów wbudowanych:

- `null`,
- `undefined`,
- `boolean`,
- `number`,
- `string`,
- `object`,
- `symbol` (dodany do języka ECMAScript 6!).

Wszystkie powyższe typy z wyjątkiem typu `object` są nazywane „prymitywnymi”.

Operator `typeof` sprawdza typ danej wartości i zawsze zwraca jedną z siedmiu wartości łańcuchowych. Co zaskakujące, w przypadku wcześniej wyszczególnionych siedmiu typów wbudowanych nie występuje dokładne dopasowanie „jeden do jednego”:

```
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";    // true
typeof { life: 42 }  === "object";    // true

// dodane w języku ECMAScript 6!
typeof Symbol()      === "symbol";    // true
```

Z sześcioma powyższymi typami powiązane są odpowiednie wartości. Jak widać, typy zwracają wartości łańcuchowe, które są nazwami tych typów. `Symbol` to nowy typ danych, który pojawił się w języku ECMAScript 6. Typ ten zostanie omówiony w rozdziale 3.

Jak być może zauważyłeś, powyższy listing nie uwzględnia typu `null`. Jest to typ *specjalny* w tym sensie, że w połączeniu z operatorem `typeof` powoduje błędne działanie:

```
typeof null === "object"; // true
```

Byłoby miło (i poprawnie!), gdyby w tym przypadku został zwrócony wynik `"null"`, ale ten istniejący od początku w języku JavaScript błąd przetrwał przez niemal dwie dekady. Prawdopodobnie nigdy nie zostanie usunięty, ponieważ istnieje tak duża liczba danych internetowych, które bazują na takim błędnym działaniu, że „naprawienie" błędu spowodowałoby wystąpienie większej liczby „błędów" i awarię większości oprogramowania internetowego.

Aby sprawdzić wartość używającą swojego typu pod kątem typu `null`, musisz skorzystać z warunku złożonego:

```
var a = null;
(!a && typeof a === "object"); // true
```

`null` to jedyny typ prymitywny, który jest „fałszywy" (inaczej mówiąc, przypominający wartość `false`; zajrzyj do rozdziału 4.), ale zwracający też `"object"` w wyniku sprawdzenia za pomocą operatora `typeof`.

Jaka zatem jest siódma wartość łańcuchowa, którą może zwrócić operator `typeof`?

```
typeof function a(){ /* .. */ } === "function"; // true
```

Łatwo pomyśleć, że słowo `function` będzie reprezentowało w języku JavaScript typ wbudowany najwyższego poziomu, zwłaszcza biorąc pod uwagę takie zachowanie operatora `typeof`. Jeśli jednak przeczytasz specyfikację, zauważysz, że `function` to w rzeczywistości swego rodzaju „podtyp" typu `object`. Dokładniej rzecz ujmując, funkcja jest określana mianem „obiektu wywoływalnego", czyli obiektu z wewnętrzną właściwością `[[Call]]`, która umożliwia jego wywołanie.

To, że funkcje to tak naprawdę obiekty, jest dość przydatne. I co najważniejsze, obiekty te są wyposażone we właściwości. Oto przykład:

```
function a(b,c) {
    /* .. */
}
```

Obiekt funkcji ma właściwość `length` służącą do ustawienia wielu formalnych parametrów, które są deklarowane wraz z nią:

```
a.length; // 2
```

Ponieważ zadeklarowano funkcję z dwoma formalnymi parametrami z nazwą (`b` i `c`), „długość funkcji" wynosi `2`.

A co z tablicami? Są one wbudowane w język JavaScript czy reprezentują specjalny typ?

```
typeof null === "object"; // true
```

W żadnym razie. Są to tylko obiekty. Jak najbardziej odpowiednie jest postrzeganie ich też jako „podtypu" typu `object` (zajrzyj do rozdziału 3.). W tym przypadku tablice cechują się dodatkowo indeksowaniem liczbowym (w przeciwieństwie do zwykłych obiektów, które bazują na stosowaniu kluczy w postaci łańcucha) oraz utrzymywaniem automatycznie aktualizowanej właściwości `.length`.

Wartości jako typy

W języku JavaScript zmienne są pozbawione typów — to *wartości mają typy*. W dowolnym momencie zmienne mogą przechowywać dowolną wartość.

Inny sposób postrzegania typów języka JavaScript polega na tym, że nie zachodzi w nim „wymuszanie typów". Oznacza to, że silnik nie wymaga, aby *zmienna* zawsze przechowywała wartości *tego samego typu początkowego*, który został użyty na początku. W ramach jednej instrukcji przypisania zmienna może przechowywać typ `string`, w następnej instrukcji może zawierać typ `number` itd.

Wartość `42` jest macierzystego typu `number`. *Typ* ten nie może zostać zmieniony. Inna wartość, taka jak `"42"` typu `string`, może zostać utworzona *na bazie* wartości `42` typu `number` z wykorzystaniem procesu nazywanego *konwersją typów* (ang. *coercion*); zajrzyj do rozdziału 4.

Użycie operatora `typeof` dla zmiennej nie oznacza wbrew temu, co może się wydawać, zadania pytania: „Jaki jest typ zmiennej?". Wynika to stąd, że zmienne w języku JavaScript nie mają typów. Formułowane jest natomiast następujące pytanie: „Jaki jest typ wartości *w* zmiennej?".

```
var a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"
```

Operator `typeof` zawsze zwraca łańcuch. A zatem:

```
typeof typeof 42; // "string"
```

W pierwszym przypadku (`typeof 42`) operator `typeof` zwraca `"number"`, a w drugim (`typeof "number"`) zwraca `"string"`.

Porównanie typu undefined z terminem „niezadeklarowane"

Zmienne, które *w danym momencie* nie mają żadnej wartości, w rzeczywistości zawierają wartość typu `undefined`. Wywołanie operatora `typeof` dla takich zmiennych spowoduje zwrócenie `"undefined"`:

```
var a;

typeof a; // "undefined"

var b = 42;
var c;
```

```
// w dalszej części kodu
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

Dla większości projektantów kuszące może być potraktowanie słowa „undefined” (niezdefiniowane) jako synonimu słowa „undeclared” (niezadeklarowane). W języku JavaScript te dwa pojęcia są jednak całkiem odmienne.

Zmienna „niezdefiniowana” to zmienna, która została zadeklarowana w dostępnym zasięgu, ale *w danym momencie* nie przechowuje żadnej wartości. Z kolei zmienna „niezadeklarowana” to zmienna, która formalnie nie została zadeklarowana w dostępnym zasięgu.

Przeanalizuj następujący kod:

```
var a;

a; // undefined
b; // błąd ReferenceError: zmienna b nie została zdefiniowana
```

Irytującą niejasnością jest komunikat o błędzie przypisywany przez przeglądarki powyższemu warunkowi. Jak widać, komunikat ma treść: „zmienna `b` nie została zdefiniowana” („`b` is not defined”). Oczywiście może on bardzo łatwo i w uzasadniony sposób zostać pomylony z komunikatem: „zmienna `b` jest typu `undefined`” („`b` is `undefined`”). Powtórzmy — w języku angielskim sformułowania „undefined” („typ `undefined`”) i „is not defined” („nie została zdefiniowana”) mają zupełnie różne znaczenia. Aby zminimalizować niejasności, byłoby miło, gdyby przeglądarki wyświetlały takie anglojęzyczne komunikaty, jak „`b` is not found” („zmienna `b` nie została znaleziona”) lub „`b` is not declared” („zmienna `b` nie została zadeklarowana”)!

Z operatorem `typeof` powiązane jest też specjalne zachowanie odnoszące się do niezadeklarowanych zmiennych, które wywołuje jeszcze większe niejasności.

Przeanalizuj następujący kod:

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

Operator `typeof` zwraca `"undefined"` nawet w przypadku „niezadeklarowanych” (lub „niezdefiniowanych”) zmiennych. Zauważ, że po wykonaniu kodu `typeof b` nie został zgłoszony żaden błąd, nawet pomimo tego, że zmienna `b` jest niezadeklarowana. Jest to specjalne zabezpieczenie związane z zachowaniem operatora `typeof`.

Podobnie jak wcześniej, byłoby miło, gdyby operator `typeof` zastosowany wraz z niezadeklarowaną zmienną spowodował zwrócenie komunikatu ze słowem „undeclared” („niezadeklarowana”), zamiast łączyć wartość wynikową z różnymi wariantami słowa „undefined” („niezdefiniowana” lub „typ `undefined`”).

Sprawdzanie pod kątem niezadeklarowanej zmiennej za pomocą operatora typeof

Mimo wszystko opisane powyżej zabezpieczenie jest przydatną możliwością przy korzystaniu z kodu JavaScript w przeglądarce, gdy wiele plików skryptowych może ładować zmienne do współużytkowanej, globalnej przestrzeni nazw.

Wielu projektantów jest przekonanych, że w globalnej przestrzeni nazw nigdy nie powinno być żadnych zmiennych, a ponadto, że wszystko powinno być zawarte w modułach i prywatnych (osobnych przestrzeniach nazw). W teorii brzmi to wspaniale, ale w praktyce jest prawie niemożliwe. Niemniej jednak jest to cel, do którego warto dążyć! Na szczęście w języku ECMAScript 6 dodano znakomitą obsługę modułów, co ostatecznie zapewni znacznie większą praktyczność.

W ramach prostego przykładu wyobraź sobie użycie „trybu debugowania" w programie, który jest kontrolowany przez zmienną globalną (flagę) o nazwie `DEBUG`. Chciałbyś sprawdzić, czy ta zmienna została zadeklarowana przed wykonaniem zadania debugowania, takiego jak rejestrowanie komunikatu w konsoli. Deklaracja zmiennej globalnej najwyższego poziomu `var DEBUG = true` zostałaby dołączona tylko do pliku *debug.js*, który ładowany jest w przeglądarce wyłącznie w trybie projektowania (testowania), lecz nie w środowisku produkcyjnym.

Musisz jednak zadbać o to, jak zmienna globalna `DEBUG` zostanie sprawdzona w reszcie kodu aplikacji, aby nie został zgłoszony błąd `ReferenceError`. W tym przypadku z pomocą przychodzi zabezpieczenie powiązane z operatorem `typeof`:

```
// Ojej, to spowodowałoby zgłoszenie błędu!
if (DEBUG) {
    console.log( "Debugowanie rozpoczyna się" );
}

// To jest bezpieczny sposób sprawdzania istnienia zmiennej.
if (typeof DEBUG !== "undefined") {
    console.log( "Debugowanie rozpoczyna się" );
}
```

Tego rodzaju sprawdzenie przydaje się nawet wtedy, gdy nie masz do czynienia ze zmiennymi zdefiniowanymi przez użytkownika (np. `DEBUG`). Jeśli sprawdzasz funkcję na potrzeby wbudowanego interfejsu API, możesz również uznać za pomocny wariant sprawdzenia bez zgłaszania błędu:

```
if (typeof atob === "undefined") {
    atob = function() { /*..*/ };
}
```

Jeśli definiujesz „uzupełniacz" (ang. *polyfill*) funkcji, która jeszcze nie istnieje, prawdopodobnie chcesz uniknąć użycia słowa kluczowego `var` do zadeklarowania zmiennej `atob`. Jeśli zmienną tę deklarujesz wewnątrz instrukcji `if` za pomocą instrukcji `var atob`, deklaracja ta jest „wynoszona" (zajrzyj do książki *Tajniki języka JavaScript. Zakresy i domknięcia* z tej samej serii) na początek zasięgu nawet wtedy, gdy nie zostanie spełniony warunek instrukcji `if` (ponieważ istnieje już zmienna globalna `atob`!). W niektórych przeglądarkach oraz w przypadku wybranych specjalnych typów wbudowanych zmiennych globalnych (często nazywanych „obiektami hosta") taka zduplikowana deklaracja może spowodować zgłoszenie błędu. Ominięcie słowa kluczowego `var` zapobiega takiej „wynoszonej" deklaracji.

Innym sposobem wykonywania takich sprawdzeń w odniesieniu do zmiennych globalnych, lecz bez funkcji zabezpieczenia operatora `typeof`, jest ustalenie, że wszystkie takie zmienne są też właściwościami obiektu globalnego, który w przeglądarce jest zasadniczo obiektem `window`. A zatem powyższe sprawdzenia mogłyby zostać przeprowadzone (dość bezpiecznie) następująco:

```
if (window.DEBUG) {
    // ..
}
if (!window.atob) {
    // ..
}
```

Inaczej, niż jest przy odwoływaniu się do niezadeklarowanych zmiennych — przy próbie uzyskania dostępu do właściwości obiektu (nawet w przypadku obiektu globalnego `window`), która nie istnieje, nie jest zgłaszany błąd `ReferenceError`.

Z kolei ręczne odwoływanie się do zmiennej globalnej za pomocą odwołania `window` jest czymś, czego niektórzy projektanci wolą unikać, zwłaszcza gdy kod musi być uruchamiany w wielu środowiskach z kodem JavaScript (nie tylko przeglądarki, ale też na przykład serwerowe środowisko node.js), w których zmienna globalna nie zawsze może mieć nazwę `window`.

Z technicznego punktu widzenia takie zabezpieczenie operatora `typeof` przydaje się nawet wtedy, gdy nie są używane zmienne globalne, choć takie sytuacje są rzadsze, a część projektantów może uznać taki wariant projektowania za mniej pożądany. Wyobraź sobie funkcję narzędziową, która ma być kopiowana i wklejana przez inne osoby do ich programów lub modułów. W funkcji ma być sprawdzane, czy program dołączający zdefiniował określoną zmienną (aby można było z niej skorzystać):

```
function doSomethingCool() {
    var helper =
        (typeof FeatureXYZ !== "undefined") ?
        FeatureXYZ :
        function() { /*.. funkcja domyślna ..*/ };
    var val = helper();
    // ..
}
```

Funkcja `doSomethingCool()` sprawdza zmienną o nazwie `FeatureXYZ`. Jeśli ją znajdzie, zostanie ona użyta. W przeciwnym razie funkcja zastosuje własną zmienną. Jeśli ktoś dołączy tę funkcję do własnego modułu (programu), w bezpieczny sposób sprawdzi ona, czy została zdefiniowana zmienna `FeatureXYZ`:

```
// wzorzec IIFE (Immediately Invoked Function Expressions; więcej na temat natychmiast wywoływanych wyrażeń funkcji
// zamieszczono w odpowiednim podrozdziale książki "Scope & Closures" z tej serii,
// w której opisano zasięgi i domknięcia)
(function(){
    function FeatureXYZ() { /*.. moja funkcja XYZ ..*/ }

    // dołączenie funkcji doSomethingCool(..)
    function doSomethingCool() {
        var helper =
            (typeof FeatureXYZ !== "undefined") ?
            FeatureXYZ :
            function() { /*.. funkcja domyślna ..*/ };
```

```
        var val = helper();
        // ..
    }

    doSomethingCool();
})();
```

W kodzie zmienna `FeatureXYZ` wcale nie jest zmienną globalną, ale jednak używane jest zabezpieczenie operatora `typeof` w celu zapewnienia bezpiecznego sprawdzenia pod kątem takiej zmiennej. Co ważne, w tym przypadku nie jest dostępny *żaden* obiekt, który można by zastosować (jakby miało to miejsce przy zmiennych globalnych, gdyby zostało użyte odwołanie `window.___`) do przeprowadzenia sprawdzenia, dlatego operator `typeof` okazuje się dość pomocny.

Część projektantów będzie preferować wzorzec projektowy określany mianem „wstrzykiwania zależności", w którym zamiast funkcji `doSomethingCool()` sprawdzającej niejawnie, czy zmienna `FeatureXYZ` ma zostać zdefiniowana poza jej obrębem, będzie wymagane jawne przekazanie zależności w następujący sposób:

```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*.. funkcja domyślna ..*/ };

    var val = helper();
// ..
}
```

Przy tworzeniu takiej funkcjonalności dostępnych jest wiele opcji. Żaden wzorzec nie jest tutaj „poprawny" ani „niepoprawny". Z każdym wariantem związane są różne kompromisy. Ogólnie jednak miłe jest to, że zabezpieczenie operatora `typeof` oferuje więcej możliwości.

Podsumowanie

W języku JavaScript dostępnych jest siedem *typów* wbudowanych: `null`, `undefined`, `boolean`, `number`, `string`, `object` i `symbol`. Mogą one być identyfikowane za pomocą operatora `typeof`.

Zmienne nie mają typów, ale zawarte w nich wartości już tak. Typy te definiują natywne zachowanie wartości.

Wielu projektantów będzie przyjmować, że terminy „niezdefiniowana" (ang. *undefined*) i „niezadeklarowana" znaczą mniej więcej to samo. W języku JavaScript są to jednak dość odmienne pojęcia. Wartość typu `undefined` może być przechowywana przez zadeklarowaną zmienną. Termin „niezadeklarowana" oznacza zmienną, która nigdy nie została zadeklarowana.

Niestety, w języku JavaScript w pewnym stopniu te dwa terminy są łączone, nie tylko w komunikatach o błędach (np. „Błąd ReferenceError: zmienna b nie została zdefiniowana"), ale też w wartościach zwracanych operatora `typeof`, który w przypadku obu terminów przekazuje `"undefined"`.

Zabezpieczenie (zapobiegające błędowi) operatora `typeof` użyte w odniesieniu do niezadeklarowanej zmiennej może być jednak pomocne w niektórych sytuacjach.

Skorowidz

N

O

P

R

S, Ś

T

U

W

Z

ZROZUM MECHANIZMY JĘZYKA JS
PISZ KOD DOBREJ JAKOŚCI!

Nawet początkujący programista może szybko zacząć tworzyć funkcjonalne aplikacje w JavaScripcie. Jest to prosty i łatwy w użyciu język, który cieszy się dużym uznaniem, a jednocześnie stanowi złożoną kolekcję mechanizmów zapożyczonych z różnych języków programowania (są to np. podstawy proceduralne znane z języka C czy podstawy funkcjonalne w stylu języka Scheme/Lisp). Bez starannej analizy mechanizmy te nie będą zrozumiałe nawet dla najbardziej doświadczonych projektantów. A przecież umiejętność rozwiązywania problemów i tworzenia kodu o naprawdę wysokiej klasie w prosty sposób zależy od takiego właśnie dogłębnego zrozumienia mechanizmów języka, w którym się pisze.

Niniejsza książka jest czwartą częścią serii w całości poświęconej językowi JavaScript. Jest przeznaczona dla osób, które używają JS w pracy i chcą dogłębnie poznać jego składniki. Omówiono tu rodzaje i zastosowanie typów oraz istotne niuanse składni. Poza ogólnymi informacjami szczegółowo opisano m.in. typy wbudowane, konwersję typów, wartości specjalne, obiekty macierzyste, prototypy macierzyste, instrukcje i wyrażenia, reguły kontekstowe. Co najważniejsze, materiał jest przedstawiony w sposób przystępny, zwięzły, klarowny i zarazem na bardzo wysokim poziomie.

Dzięki tej książce:

- poznasz siedem typów języka JavaScript: *null*, *undefined*, *boolean*, *number*, *string*, *object* i *symbol*
- nabierzesz biegłości w programowaniu asynchronicznym w języku JavaScript
- nauczysz się stosować obietnice JavaScript i wykorzystasz je do pisania asynchronicznych API
- będziesz wykorzystywać generatory do wyrażania asynchroniczności w sposób sekwencyjny i wyglądający na synchroniczny
- dowiesz się, w jaki sposób zoptymalizować wydajność na poziomie programu za pomocą wątków roboczych, SIMD i stylu asm.js
- poznasz nieocenione zasoby i techniki przeznaczone do przeprowadzania testów jednostkowych oraz dostrajania wyrażeń i poleceń

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. Pisze książki, prowadzi warsztaty, występuje na konferencjach o tematyce technicznej oraz pozostaje aktywnym członkiem społeczności OSS.

Helion

42357 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/nowosci

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

Informatyka w najlepszym wydaniu

ISBN 978-83-283-2174-8

9 788328 321748

cena: 34,90 zł

O'REILLY