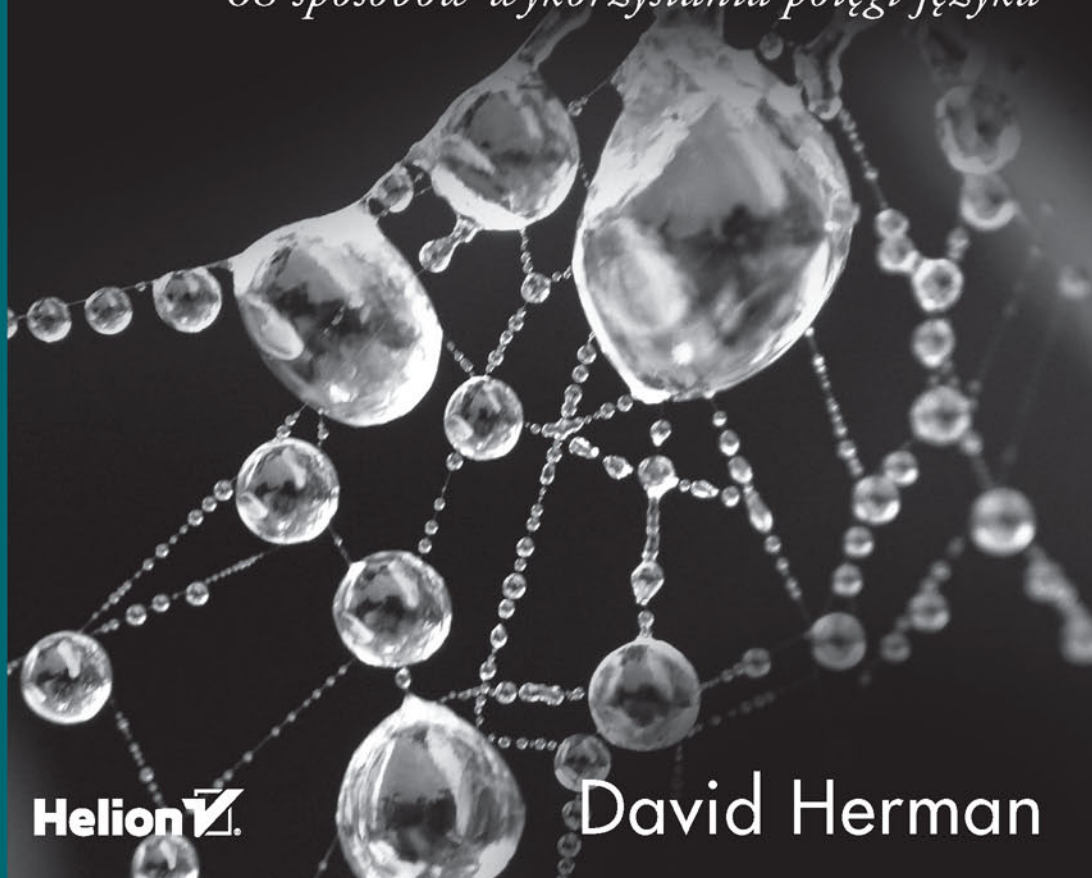




Efektowny JAVASCRIPT

68 sposobów wykorzystania potęgi języka



Helion 

David Herman

Tytuł oryginału: Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-1418-4

Authorized translation from the English language edition, entitled: EFFECTIVE JAVASCRIPT: 68 SPECIFIC WAYS TO HARNESS THE POWER OF JAVASCRIPT; ISBN 0321812182; by David Herman; published by Pearson Education, Inc, publishing as Addison Wesley.
Copyright © 2013 by Pearson Education, Inc.

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 Pearson Education, Inc.

Polish language edition published by HELION S.A. Copyright © 2015.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/efprjs.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/efprjs>

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	11
Wprowadzenie	13
Podziękowania	15
O autorze	17
Rozdział 1. Przyzwyczajanie się do JavaScriptu	19
Sposób 1. Ustal, której wersji JavaScriptu używasz.....	19
Sposób 2. Liczby zmiennoprzecinkowe w JavaScriptcie.....	24
Sposób 3. Uważaj na niejawną konwersję typu.....	27
Sposób 4. Stosuj typy proste zamiast nakładek obiektowych.....	32
Sposób 5. Unikaj stosowania operatora == dla wartości o różnych typach	34
Sposób 6. Ograniczenia mechanizmu automatycznego dodawania średników	37
Sposób 7. Traktuj łańcuchy znaków jak sekwencje 16-bitowych jednostek kodowych.....	43
Rozdział 2. Zasięg zmiennych	47
Sposób 8. Minimalizuj liczbę obiektów globalnych.....	47
Sposób 9. Zawsze deklaruuj zmienne lokalne	50
Sposób 10. Unikaj słowa kluczowego with.....	51
Sposób 11. Poznaj domknięcia	54
Sposób 12. Niejawne przenoszenie deklaracji zmiennych na początek bloku (czyli hoisting).....	57
Sposób 13. Stosuj wyrażenia IIFE do tworzenia zasięgu lokalnego	59

Sposób 14. Uważaj na nieprzenośne określanie zasięgu nazwanych wyrażeń funkcyjnych.....	62
Sposób 15. Uważaj na nieprzenośne określanie zasięgu lokalnych deklaracji funkcji w bloku	65
Sposób 16. Unikaj tworzenia zmiennych lokalnych za pomocą funkcji eval	67
Sposób 17. Przedkładaj pośrednie wywołania eval nad bezpośrednie wywołania tej funkcji	68

Rozdział 3. Korzystanie z funkcji..... 71

Sposób 18. Różnice między wywołaniami funkcji, metod i konstruktorów.....	71
Sposób 19. Funkcje wyższego poziomu	74
Sposób 20. Stosuj instrukcję call do wywoływania metod dla niestandardowego odbiorcy.....	77
Sposób 21. Stosuj instrukcję apply do wywoływania funkcji o różnej liczbie argumentów.....	79
Sposób 22. Stosuj słowo kluczowe arguments do tworzenia funkcji wariadycznych.....	81
Sposób 23. Nigdy nie modyfikuj obiektu arguments	82
Sposób 24. Używaj zmiennych do zapisywania referencji do obiektu arguments.....	84
Sposób 25. Używaj instrukcji bind do pobierania metod o stałym odbiorcy...	85
Sposób 26. Używaj metody bind do wiązania funkcji z podzbiorem argumentów (technika currying)	87
Sposób 27. Wybieraj domknięcia zamiast łańcuchów znaków do hermetyzowania kodu	88
Sposób 28. Unikaj stosowania metody toString funkcji	90
Sposób 29. Unikaj niestandardowych właściwości przeznaczonych do inspekcji stosu.....	92

Rozdział 4. Obiekty i prototypy 95

Sposób 30. Różnice między instrukcjami prototype, getPrototypeOf i __proto__.....	95
Sposób 31. Stosuj instrukcję Object.getPrototypeOf zamiast __proto__	99
Sposób 32. Nigdy nie modyfikuj właściwości __proto__	100
Sposób 33. Uniezależnianie konstruktorów od instrukcji new	101
Sposób 34. Umieszczaj metody w prototypach.....	103
Sposób 35. Stosuj domknięcia do przechowywania prywatnych danych	105
Sposób 36. Stan egzemplarzy przechowuj tylko w nich samych.....	107
Sposób 37. Zwracaj uwagę na niejawne wiązanie obiektu this.....	109

Sposób 38. Wywoływanie konstruktorów klasy bazowej w konstruktorach klas pochodnych	111
Sposób 39. Nigdy nie wykorzystuj ponownie nazw właściwości z klasy bazowej.....	115
Sposób 40. Unikaj dziedziczenia po klasach standardowych.....	117
Sposób 41. Traktuj prototypy jak szczegół implementacji	119
Sposób 42. Unikaj nieprzemyślanego stosowania techniki monkey patching	120
Rozdział 5. Tablice i słowniki	123
Sposób 43. Budowanie prostych słowników na podstawie egzemplarzy typu Object.....	123
Sposób 44. Stosuj prototypy null, aby uniknąć zaśmiecania przez prototypy	126
Sposób 45. Używaj metody hasOwnProperty do zabezpieczania się przed zaśmiecaniem przez prototypy	
Sposób 46. Stosuj tablice zamiast słowników przy tworzeniu kolekcji uporządkowanych	132
Sposób 47. Nigdy nie dodawaj enumerowanych właściwości do prototypu Object.prototype	134
Sposób 48. Unikaj modyfikowania obiektu w trakcie enumeracji.....	136
Sposób 49. Stosuj pętlę for zamiast pętli for...in przy przechodzeniu po tablicy	140
Sposób 50. Zamiast pętli stosuj metody do obsługi iteracji	142
Sposób 51. Wykorzystaj uniwersalne metody klasy Array w obiektach podobnych do tablic	146
Sposób 52. Przedkładaj literały tablicowe nad konstruktor klasy Array	148
Rozdział 6. Projekty bibliotek i interfejsów API	151
Sposób 53. Przestrzegaj spójnych konwencji	151
Sposób 54. Traktuj wartość undefined jak brak wartości.....	153
Sposób 55. Stosuj obiekty z opcjami do przekazywania argumentów za pomocą słów kluczowych.....	157
Sposób 56. Unikaj niepotrzebnego przechowywania stanu	161
Sposób 57. Określaj typy na podstawie struktury, aby stworzyć elastyczne interfejsy.....	164
Sposób 58. Różnice między tablicami a obiektami podobnymi do tablic	167
Sposób 59. Unikaj nadmiernej koercji.....	171
Sposób 60. Obsługa łańcuchów metod	174

Rozdział 7. Współbieżność	179
Sposób 61. Nie blokuj kolejki zdarzeń operacjami wejścia-wyjścia	180
Sposób 62. Stosuj zagnieżdżone lub nazwane wywołania zwrotne do tworzenia sekwencji asynchronicznych wywołań	183
Sposób 63. Pamiętaj o ignorowanych błędach	187
Sposób 64. Stosuj rekurencję do tworzenia asynchronicznych pętli	190
Sposób 65. Nie blokuj kolejki zdarzeń obliczeniami	193
Sposób 66. Wykorzystaj licznik do wykonywania współbieżnych operacji ...	197
Sposób 67. Nigdy nie uruchamiaj synchronicznie asynchronicznych wywołań zwrotnych	201
Sposób 68. Stosuj obietnice, aby zwiększyć przejrzystość asynchronicznego kodu	203
Skorowidz	207

3

Korzystanie z funkcji

Funkcje to „woły robocze” JavaScriptu. Są dla programistów jednocześnie podstawową abstrakcją i mechanizmem implementacyjnym. Funkcje odgrywają tu rolę, które w innych językach są przypisane do wielu różnych elementów: procedur, metod, konstruktorów, a nawet klas i modułów. Gdy zapoznasz się z subtelnymi aspektami funkcji, opanujesz istotną część JavaScriptu. Pamiętaj jednak o tym, że nauka efektywnego posługiwania się funkcjami w różnych kontekstach wymaga czasu.

Sposób 18. Różnice między wywołaniami funkcji, metod i konstruktorów

Jeśli programowanie obiektowe nie jest Ci obce, prawdopodobnie traktujesz funkcje, metody i konstruktory klas jako trzy odrębne elementy. W JavaScriptcie odpowiadają im trzy różne sposoby korzystania z jednej konstrukcji — z funkcji.

Najprostszy sposób to wywołanie funkcji:

```
function hello(username) {  
    return "Witaj, " + username;  
}  
hello("Keyser Söze"); // "Witaj, Keyser Söze"
```

Ten kod działa w standardowy sposób — wywołuje funkcję `hello` i wiąże parametr `name` z podanym argumentem.

Metody w JavaScriptcie to należące do obiektów właściwości, które są funkcjami:

```
var obj = {  
    hello: function() {  
        return "Witaj, " + this.username;  
    }  
};
```

```
    username: "Hans Gruber"  
};  
obj.hello(); // "Witaj, Hans Gruber"
```

Zauważ, że w metodzie `hello` używane jest słowo `this`, aby uzyskać dostęp do obiektu `obj`. Może się wydawać, że `this` zostaje związane z `obj`, ponieważ metodę `hello` zdefiniowano właśnie w obiekcie `obj`. Jednak można skopiować referencję do tej samej funkcji w innym obiekcie i uzyskać odmienny wynik:

```
var obj2 = {  
    hello: obj.hello,  
    username: "Boo Radley"  
};  
obj2.hello(); // "Witaj, Boo Radley"
```

W wywołaniu metody samo wywołanie określa, z czym związane jest słowo `this` (wyznacza ono odbiorcę wywołania). Wyrażenie `obj.hello()` powoduje wyszukanie właściwości `hello` obiektu `obj` i wywołanie jej dla odbiorcy `obj`. Wyrażenie `obj2.hello()` prowadzi do wyszukiwania właściwości `hello` obiektu `obj2`; tą właściwością jest ta sama funkcja co w wywołaniu `obj.hello`, tu jednak jest ona wywoływana dla odbiorcy `obj2`. Wywołanie metody obiektu standardowo prowadzi do wyszukania metody i użycia danego obiektu jako odbiorcy tej metody.

Ponieważ metody to funkcje wywołane dla określonego obiektu, można swobodnie wywoływać zwykle funkcje z wykorzystaniem słowa kluczowego `this`:

```
function hello() {  
    return "Witaj, " + this.username;  
}
```

Takie rozwiązanie może okazać się przydatne, jeśli chcesz wstępnie zdefiniować funkcję, która będzie używana w wielu obiektach:

```
var obj1 = {  
    hello: hello,  
    username: "Gordon Gekko"  
};  
obj1.hello(); // "Witaj, Gordon Gekko"  
  
var obj2 = {  
    hello: hello,  
    username: "Biff Tannen"  
};  
obj2.hello(); // "Witaj, Biff Tannen"
```

Jednak funkcja wykorzystująca słowo kluczowe `this` nie jest przydatna, jeśli chcesz ją wywoływać jak zwykłą funkcję, a nie jak metodę:

```
hello(); // "Witaj, undefined"
```

Nie pomaga to, że w zwykłych wywołaniach funkcji (nie w metodach) odbiorcą jest obiekt globalny, który tu nie ma właściwości o nazwie `name` i zwraca wartość `undefined`. Wywołanie metody jako funkcji rzadko jest przydatne, jeśli

dana metoda wykorzystuje słowo kluczowe `this`. Przyczyną jest to, że trudno oczekiwać, iż obiekt globalny będzie miał te same właściwości co obiekt, dla którego napisano daną metodę. Używanie w tym kontekście obiektu globalnego jest na tyle problematycznym rozwiązaniem domyślnym, że w trybie `strict` w standardzie ES5 `this` jest domyślnie wiązane z wartością `undefined`:

```
function hello() {
  "use strict";
  return "Witaj, " + this.username;
}
hello(); // Błąd: nie można znaleźć właściwości "username" obiektu undefined
```

To pomaga wykryć przypadkowe wykorzystanie metod jako zwykłych funkcji. Kod szybko przestanie wtedy działać, ponieważ próba dostępu do właściwości obiektu `undefined` spowoduje natychmiastowe zgłoszenie błędu.

Trzecim sposobem używania funkcji jest ich wywoływanie jako konstruktorów. Konstruktory, podobnie jak metody i zwykłe funkcje, definiuje się za pomocą słowa kluczowego `function`:

```
function User(name, passwordHash) {
  this.name = name;
  this.passwordHash = passwordHash;
}
```

Jeśli wywołasz funkcję `User` z operatorem `new`, zostanie ona potraktowana jak konstruktor:

```
var u = new User("sfalken",
  "0ef33ae791068ec64b502d6cb0191387");
u.name; // "sfalken"
```

Wywołanie konstruktora, w odróżnieniu od wywołań funkcji i metod, powoduje przekazanie nowego obiektu jako wartości `this` i niejawne zwrócenie nowego obiektu jako wyniku. Głównym zadaniem konstruktora jest inicjowanie obiektów.

Co warto zapamiętać?

- W wywołaniach metod należy podać obiekt (odbiorcę), w którym szukana będzie właściwość w postaci tej metody.
- W wywołaniach funkcji odbiorcą jest obiekt globalny (w trybie `strict` jest to wartość `undefined`). Wywoływanie metod jak zwykłych funkcji rzadko jest przydatne.
- Konstruktory są wywoływane za pomocą słowa kluczowego `new`, a ich odbiorcą są nowe obiekty.

Sposób 19. Funkcje wyższego poziomu

Funkcje wyższego poziomu były w przeszłości znakiem rozpoznawczym mistrzów programowania funkcyjnego. Te tajemnicze nazwy sugerują, że mamy tu do czynienia z zaawansowaną techniką programistyczną. Nic bardziej mylnego. Wykorzystanie zwięzłej elegancji funkcji często prowadzi do powstania prostszego i krótszego kodu. Przez lata w językach skryptowych wprowadzono techniki z tego obszaru. Dzięki temu udało się przybliżyć użytkownikom niektóre z najlepszych idiomów z dziedziny programowania funkcyjnego.

Funkcje wyższego poziomu to po prostu funkcje, które przyjmują inne funkcje jako argumenty lub zwracają inne funkcje jako wynik. Zwłaszcza przyjmowanie argumentu w postaci funkcji (nazywanej często **funkcją wywoływaną zwrrotnie**, ponieważ jest wywoływana przez funkcję wyższego poziomu) to wyjątkowo przydatny i zwięzły idiom, często wykorzystywany w programach w JavaScriptcie.

Przyjrzyj się standardowej metodzie `sort` tablic. Aby mogła ona działać dla wszystkich możliwych tablic, wymaga od programu wywołującego określenia, jak należy porównywać dwa elementy tablicy:

```
function compareNumbers(x, y) {
  if (x < y) {
    return -1;
  }
  if (x > y) {
    return 1;
  }
  return 0;
}
[3, 1, 4, 1, 5, 9].sort(compareNumbers); // [1, 1, 3, 4, 5, 9]
```

W bibliotece standardowej można było zażądać, aby program wywołujący przekazywał obiekt z metodą `compare`, jednak ponieważ wymagana jest tylko jedna metoda, bezpośrednie przyjmowanie funkcji to prostsze i bardziej zwięzłe rozwiązanie. Przedstawiony wcześniej przykład można uprościć jeszcze bardziej, stosując funkcję anonimową:

```
[3, 1, 4, 1, 5, 9].sort(function(x, y) {
  if (x < y) {
    return -1;
  }
  if (x > y) {
    return 1;
  }
  return 0;
}); // [1, 1, 3, 4, 5, 9]
```

Opanowanie funkcji wyższego poziomu często pozwala uprościć kod i wyeliminować nudny, szablonowy kod. Dla wielu typowych operacji na tablicach istnieją świetne abstrakcje wyższego rzędu, z którymi warto się zapoznać.

Przyjrzyj się prostemu zadaniu przekształcania tablicy łańcuchów znaków. Za pomocą pętli można napisać następujący kod:

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = [];
for (var i = 0, n = names.length; i < n; i++) {
    upper[i] = names[i].toUpperCase();
}
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Dzięki wygodnej metodzie `map` tablic (wprowadzonej w standardzie ES5) można całkowicie zrezygnować z pętli i zaimplementować transformację kolejnych elementów za pomocą funkcji lokalnej:

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = names.map(function(name) {
    return name.toUpperCase();
});
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Gdy przyzwyczaisz się do korzystania z funkcji wyższego poziomu, zaczniesz dostrzegać okazje do ich samodzielnego pisania. Dobrą oznaką wskazującą, że taka funkcja może okazać się przydatna, jest występowanie powtarzającego się lub podobnego kodu. Załóżmy, że jedna część programu tworzy łańcuch znaków składający się z liter alfabetu:

```
var aIndex = "a".charCodeAt(0); // 97

var alphabet = "";
for (var i = 0; i < 26; i++) {
    alphabet += String.fromCharCode(aIndex + i);
}
alphabet; // "abcdefghijklmnopqrstuvwxyz"
```

Inna część programu generuje łańcuch znaków obejmujący cyfry:

```
var digits = "";
for (var i = 0; i < 10; i++) {
    digits += i;
}
digits; // "0123456789"
```

W jeszcze innym fragmencie program tworzy łańcuch z losowych znaków:

```
var random = "";

for (var i = 0; i < 8; i++) {
    random += String.fromCharCode(Math.floor(Math.random() * 26)
    + aIndex);
}
random; // "bdwvfrtp" (za każdym razem wynik jest inny)
```

Każdy fragment generuje inny łańcuch znaków, jednak logika działania jest za każdym razem podobna. Każda z pokazanych pętli generuje łańcuch znaków na podstawie scalania wyników obliczeń dających poszczególne znaki.

Można wyodrębnić wspólne aspekty i umieścić je w jednej funkcji narzędziowej:

```
function buildString(n, callback) {
  var result = "";
  for (var i = 0; i < n; i++) {
    result += callback(i);
  }
  return result;
}
```

Zauważ, że w funkcji `buildString` znajdują się wszystkie wspólne elementy każdej pętli, natomiast dla zmiennych aspektów stosowane są parametry. Liczbie iteracji pętli odpowiada zmienna `n`, a do tworzenia każdego fragmentu łańcucha służy funkcja `callback`. Teraz można uprościć każdy z trzech przykładów i zastosować w nich funkcję `buildString`:

```
var alphabet = buildString(26, function(i) {
  return String.fromCharCode(aIndex + i);
});
alphabet; // "abcdefghijklmnopqrstuvwxyz"

var digits = buildString(10, function(i) { return i; });
digits; // "0123456789"

var random = buildString(8, function() {
  return String.fromCharCode(Math.floor(Math.random() * 26)
    + aIndex);
});
random; // "ltvisfr" (wynik za każdym razem jest inny)
```

Tworzenie abstrakcji wyższego poziomu przynosi wiele korzyści. Jeśli w implementacji występują skomplikowane fragmenty (trzeba na przykład właściwie obsłużyć warunki graniczne dla pętli), znajdują się one w funkcji wyższego poziomu. Dzięki temu wystarczy naprawić błędy w logice raz, zamiast szukać wszystkich wystąpień danego wzorca rozrzuconych po programie. Także jeśli stwierdzisz, że trzeba zoptymalizować wydajność operacji, wystarczy to zrobić w jednym miejscu. Ponadto nadanie abstrakcji jednoznacznej nazwy (takiej jak `buildString`, czyli „twórz łańcuch znaków”) jasno informuje czytelników kodu o jego działaniu. Dzięki temu nie trzeba analizować szczegółów implementacji.

Stosowanie funkcji wyższego poziomu po dostrzeżeniu, że w kodzie powtarza się ten sam wzorzec, prowadzi do powstawania bardziej zwięzłego kodu, zwiększenia produktywności i poprawy czytelności kodu. Zwracanie uwagi na powtarzające się wzorce i przenoszenie ich do funkcji narzędziowych wyższego poziomu to ważny nawyk, który warto sobie rozwinąć.

Co warto zapamiętać?

- Funkcje wyższego poziomu charakteryzują się tym, że przyjmują inne funkcje jako argumenty lub zwracają funkcje jako wyniki.
- Zapoznaj się z funkcjami wyższego poziomu z istniejących bibliotek.
- Naucz się wykrywać powtarzające się wzorce, które można zastąpić funkcjami wyższego poziomu.

Sposób 20. Stosuj instrukcję call do wywoływania metod dla niestandardowego odbiorcy

Odbiorca funkcji lub metody (czyli wartość wiązana ze specjalnym słowem kluczowym `this`) standardowo jest określany na podstawie składni wywołania. Wywołanie metody powoduje związanie ze słowem kluczowym `this` obiektu, w którym dana metoda jest wyszukiwana. Czasem jednak trzeba wywołać funkcję dla niestandardowego odbiorcy, a nie jest ona jego właściwością. Można oczywiście dodać potrzebną metodę jako nową właściwość danego obiektu:

```
obj.temporary = f; // Co się stanie, jeśli właściwość obj.temporary już istniała?  
var result = obj.temporary(arg1, arg2, arg3);  
delete obj.temporary; // Co się stanie, jeśli właściwość obj.temporary już istniała?
```

Jednak to podejście jest niewygodne, a nawet niebezpieczne. Często nie należy, a nawet nie da się zmodyfikować obiektu takiego jak `obj` w przykładzie. Niezależnie od nazwy wybranej dla właściwości (tu jest to `temporary`) istnieje ryzyko kolizji z istniejącą właściwością obiektu. Ponadto niektóre obiekty są zamrożone lub zamknięte, co uniemożliwia dodawanie do nich nowych właściwości. Ponadto dodawanie dowolnych właściwości do obiektów to zła praktyka — zwłaszcza gdy są to obiekty utworzone przez innego programistę (zobacz Sposób 42.).

Na szczęście funkcje mają wbudowaną metodę `call`, umożliwiającą podanie niestandardowego odbiorcy. Wywołanie funkcji za pomocą metody `call`:

```
f.call(obj, arg1, arg2, arg3);
```

działa podobnie jak wywołanie bezpośrednie:

```
f(arg1, arg2, arg3);
```

Różnica polega na tym, że w metodzie `call` pierwszy argument to jawnie wskazany obiekt odbiorcy.

Metoda `call` jest wygodna przy wywoływaniu metod, które mogły zostać usunięte, zmodyfikowane lub zastąpione. W sposobie 45. przedstawiony jest przydatny przykład, ilustrujący wywoływanie metody `hasOwnProperty` dla dowolnych

obiektów (nawet dla słownika). W słowniku sprawdzenie właściwości `hasOwnProperty` powoduje zwrócenie wartości ze słownika, zamiast wywołania odziedziczonej metody:

```
dict.hasOwnProperty = 1;
dict.hasOwnProperty("foo"); // Błąd: 1 nie jest funkcją
```

Za pomocą metody `call` można wywołać metodę `hasOwnProperty` dla słownika, nawet jeśli nie jest ona zapisana w samym obiekcie:

```
var hasOwnProperty = {}.hasOwnProperty;
dict.foo = 1;
delete dict.hasOwnProperty;
hasOwnProperty.call(dict, "foo"); // true
hasOwnProperty.call(dict, "hasOwnProperty"); // false
```

Metoda `call` jest przydatna także przy definiowaniu funkcji wyższego poziomu. Często stosowany idiom dotyczący funkcji wyższego poziomu polega na przyjmowaniu opcjonalnych argumentów określających odbiorcę, dla którego funkcja ma zostać wywołana. Na przykład obiekt reprezentujący tablicę z parami klucz – wartość może udostępniać metodę `forEach`:

```
var table = {
  entries: [],
  addEntry: function(key, value) {
    this.entries.push({ key: key, value: value });
  },
  forEach: function(f, thisArg) {
    var entries = this.entries;
    for (var i = 0, n = entries.length; i < n; i++) {
      var entry = entries[i];
      f.call(thisArg, entry.key, entry.value, i);
    }
  }
};
```

To umożliwi użytkownikom obiektu podanie potrzebnej metody jako wywoływanej zwrótnie funkcji `f` z metody `table.forEach` i wskazanie odpowiedniego odbiorcy. Dzięki temu można na przykład wygodnie skopiować zawartość jednej tablicy do drugiej:

```
table1.forEach(table2.addEntry, table2);
```

Ten kod używa metody `addEntry` obiektu `table2` (można też pobrać tę metodę z obiektu `Table.prototype` lub `table1`), a metoda `forEach` wielokrotnie wywołuje metodę `addEntry` dla odbiorcy `table2`. Zauważ, że choć metoda `addEntry` oczekuje tylko dwóch argumentów, metoda `forEach` wywołuje ją z trzema argumentami: kluczem, wartością i indeksem. Dodatkowy argument w postaci indeksu nie powoduje problemów, ponieważ metoda `addEntry` po prostu go pomija.

Co warto zapamiętać?

- Używaj metody `call` do wywoływania funkcji dla niestandardowych odbiorców.
- Stosuj metodę `call` do wywoływania metod, które mogą nie istnieć w danym obiekcie.
- Używaj metody `call` do definiowania funkcji wyższego poziomu, umożliwiających klientom określanie odbiorcy wywoływanych zwrótnie funkcji.

Sposób 21. Stosuj instrukcję apply do wywoływania funkcji o różnej liczbie argumentów

Wyobraź sobie, że dostępna jest funkcja obliczająca średnią z dowolnej liczby wartości:

```
average(1, 2, 3);           // 2
average(1);                 // 1
average(3, 1, 4, 1, 5, 9, 2, 6, 5); // 4
average(2, 7, 1, 8, 2, 8, 1, 8); // 4.625
```

Funkcja `average` jest funkcją **wariadyczną** (ang. *variadic*), inaczej funkcją o **zmiennej arności** (arność funkcji to liczba oczekiwanych przez nią argumentów). Oznacza to tyle, że może przyjmować dowolną liczbę argumentów. Wersja funkcji `average` mająca stałą arność prawdopodobnie pobierałaby jeden argument z tablicą wartości:

```
averageOfArray([1, 2, 3]);           // 2
averageOfArray([1]);                 // 1
averageOfArray([3, 1, 4, 1, 5, 9, 2, 6, 5]); // 4
averageOfArray([2, 7, 1, 8, 2, 8, 1, 8]); // 4.625
```

Wersja wariadyczna jest bardziej zwięzła i (choć to kwestia dyskusyjna) bardziej elegancka. Funkcje wariadyczne mają wygodną składnię — przynajmniej wtedy, gdy użytkownik od razu wie, ile argumentów chce podać. Tak dzieje się w przedstawionych przykładach. Wyobraź sobie jednak, że tablica wartości jest tworzona w następujący sposób:

```
var scores = getAllScores();
```

Jak użyć funkcji `average` do obliczenia średniej z tych wartości?

```
average(/* ? */);
```

Na szczęście funkcje mają wbudowaną metodę `apply`. Działa ona podobnie jak metoda `call`, ale jest zaprojektowana w określonym celu. Metoda `apply` przyjmuje tablicę argumentów i wywołuje daną funkcję w taki sposób, jakby każdy element tablicy był odrębnym argumentem wywołania tej funkcji. Oprócz

tablicy argumentów metoda `apply` przyjmuje dodatkowo pierwszy argument, określający obiekt `this` dla wywoływanej funkcji. Ponieważ funkcja `average` nie używa obiektu `this`, wystarczy podać wartość `null`:

```
var scores = getAllScores();
average.apply(null, scores);
```

Jeśli tablica `scores` będzie miała na przykład trzy elementy, przedstawiony kod zadziała tak samo jak poniższa wersja:

```
average(scores[0], scores[1], scores[2]);
```

Metoda `apply` działa także dla metod wariadycznych. Na przykład obiekt `buffer` może mieć wariadyczną metodę `append`, przeznaczoną do dodawania elementów do jego wewnętrznego stanu (aby zrozumieć implementację tej metody `append`, zapoznaj się ze Sposobem 22.):

```
var buffer = {
  state: [],
  append: function() {
    for (var i = 0, n = arguments.length; i < n; i++) {
      this.state.push(arguments[i]);
    }
  }
};
```

Metodę `append` można wywołać z dowolną liczbą argumentów:

```
buffer.append("Witaj, ");
buffer.append(firstName, " ", lastName, "!");
buffer.append(newline);
```

Za pomocą argumentu metody `apply` określającego obiekt `this` można też wywołać tę metodę z generowaną tablicą:

```
buffer.append.apply(buffer, getInputStrings());
```

Zwróć uwagę na znaczenie argumentu `buffer`. Jeśli przekażesz niewłaściwy obiekt, metoda `append` spróbuje zmodyfikować właściwość `state` nieodpowiedniego obiektu.

Co warto zapamiętać?

- Stosuj metodę `apply` do wywoływania funkcji wariadycznych z generowanymi tablicami argumentów.
- Za pomocą pierwszego argumentu metody `apply` możesz wskazać odbiorcę metod wariadycznych.

Sposób 22. Stosuj słowo kluczowe arguments do tworzenia funkcji wariadycznych

W sposobie 21. znajduje się opis funkcji wariadycznej `average`. Potrafi ona zwrócić średnią z dowolnej liczby argumentów. Jak samodzielnie zaimplementować funkcję wariadyczną? Napisanie funkcji `averageOfArray` o stałej arności jest stosunkowo łatwe:

```
function averageOfArray(a) {
  for (var i = 0, sum = 0, n = a.length; i < n; i++) {
    sum += a[i];
  }
  return sum / n;
}
averageOfArray([2, 7, 1, 8, 2, 8, 1, 8]); // 4.625
```

W tej definicji funkcji `averageOfArray` używany jest jeden **parametr formalny** — zmienna `a` z listy parametrów. Gdy użytkownicy wywołują funkcję `averageOfArray`, podają pojedynczy **argument** (nazywany tak w celu odróżnienia od parametru formalnego). Jest nim tablica wartości.

Wersja wariadyczna wygląda niemal tak samo, ale nie są w niej jawnie zdefiniowane żadne parametry formalne. Zamiast tego wykorzystywany jest fakt, że JavaScript dodaje do każdej funkcji niejawną zmienną lokalną o nazwie `arguments`. Obiekt `arguments` umożliwia dostęp do argumentów w sposób przypominający używanie tablicy. Każdy argument ma określony indeks, a właściwość `length` określa, ile argumentów zostało podanych. To sprawia, że w funkcji `average` o zmiennej arności można przejść w pętli po wszystkich elementach z obiektu `arguments`:

```
function average() {
  for (var i = 0, sum = 0, n = arguments.length;
    i < n; i++) {
    sum += arguments[i];
  }
  return sum / n;
}
```

Funkcje wariadyczne dają dużo swobody. W różnych miejscach można je wywoływać z wykorzystaniem odmiennej liczby argumentów. Jednak same w sobie mają pewne ograniczenia, ponieważ gdy użytkownicy chcą je wywołać dla generowanej tablicy argumentów, muszą zastosować opisaną w sposobie 21. metodę `apply`. Gdy w celu ułatwienia pracy udostępniasz funkcję o zmiennej arności, to zgodnie z ogólną regułą powinienesz też utworzyć wersję o stałej arności, przyjmującą jawnie podaną tablicę. Zwykle jest to łatwe, ponieważ przeważnie można zaimplementować funkcję wariadyczną jako prostą nakładkę przekazującą zadania do wersji o stałej arności:

```
function average() {
    return averageOfArray(arguments);
}
```

Dzięki temu użytkownicy funkcji nie muszą uciekać się do stosowania metody `apply`, która może zmniejszać czytelność kodu i często prowadzi do spadku wydajności.

Co warto zapamiętać?

- Stosuj niejawny obiekt `arguments` do implementowania funkcji o zmiennej arności.
- Pomyśl o udostępnieniu obok funkcji wariadycznych dodatkowych wersji o stałej arności, aby użytkownicy nie musieli stosować metody `apply`.

Sposób 23. Nigdy nie modyfikuj obiektu `arguments`

Obiekt `arguments` wprawdzie wygląda jak tablica, ale niestety nie zawsze działa w ten sposób. Programiści znający Perla i uniksowe skrypty powłoki są przyzwyczajeni do stosowania techniki przesuwania elementów w kierunku początku tablicy argumentów. Tablice w JavaScriptcie udostępniają metodę `shift`, która usuwa pierwszy element tablicy i przesuwa wszystkie kolejne elementy o jedną pozycję. Jednak obiekt `arguments` nie jest egzemplarzem standardowego typu `Array`, dlatego nie można bezpośrednio wywołać metody `arguments.shift()`.

Może się wydawać, że dzięki metodzie `call` da się pobrać metodę `shift` tablic i wywołać ją dla obiektu `arguments`. Na pozór jest to sensowny sposób implementacji funkcji takiej jak `callMethod`, która przyjmuje obiekt i nazwę metody oraz próbuje wywołać wskazaną metodę tego obiektu dla wszystkich pozostałych argumentów:

```
function callMethod(obj, method) {
    var shift = [].shift;
    shift.call(arguments);
    shift.call(arguments);
    return obj[method].apply(obj, arguments);
}
```

Jednak działanie tej funkcji jest dalekie od oczekiwanego:

```
var obj = {
    add: function(x, y) { return x + y; }
};
callMethod(obj, "add", 17, 25);
// Błąd: nie można wczytać właściwości "apply" obiektu undefined
```

Przyczyną problemów z tym kodem jest to, że obiekt `arguments` nie jest kopią argumentów funkcji. Wszystkie nazwane argumenty to *aliasy* odpowiednich indeksów z obiektu `arguments`. Tak więc `obj` to alias dla `arguments[0]`, a `method` to alias dla `arguments[1]`. Jest tak nawet po usunięciu elementów z obiektu `arguments` za pomocą wywołania `shift`. To oznacza, że choć na pozór używane jest wywołanie `obj["add"]`, w rzeczywistości wywołanie to `17[25]`. Na tym etapie zaczynają się kłopoty. Z powodu obowiązującej w JavaScriptcie automatycznej konwersji typów wartość `17` jest przekształcana w obiekt typu `Number`, po czym pobierana jest jego (nieistniejąca) właściwość `"25"`, dlatego zwrócona zostaje wartość `undefined`. Potem następuje nieudana próba pobrania właściwości `"apply"` obiektu `undefined` w celu wywołania jej jako metody.

Wniosek z tego jest taki, że relacja między obiektem `arguments` a nazwanymi parametrami funkcji łatwo staje się źródłem problemów. Modyfikacja obiektu `arguments` grozi przekształceniem nazwanych parametrów funkcji w bezsensowne dane. W trybie `strict` ze standardu ES5 komplikacje są jeszcze większe. Parametry funkcji w tym trybie *nie* są aliasami elementów obiektu `arguments`. Aby zademonstrować różnicę, można napisać funkcję aktualizującą element z obiektu `arguments`:

```
function strict(x) {
  "use strict";
  arguments[0] = "zmodyfikowany";
  return x === arguments[0];
}
function nonstrict(x) {
  arguments[0] = "zmodyfikowany";
  return x === arguments[0];
}
strict("niezmodyfikowany"); // false
nonstrict("niezmodyfikowany"); // true
```

Dlatego dużo bezpieczniej jest nigdy nie modyfikować obiektu `arguments`. Można łatwo uzyskać ten efekt, kopiując najpierw elementy z tego obiektu do zwykłej tablicy. Oto prosty idiom ilustrujący taką modyfikację:

```
var args = [].slice.call(arguments);
```

Metoda `slice` tablic tworzy kopię tablicy, gdy zostanie wywołana bez dodatkowych argumentów. Zwracany wynik to egzemplarz standardowego typu `Array`. Ten egzemplarz nie jest aliasem żadnych obiektów i umożliwia bezpośredni dostęp do wszystkich metod typu `Array`.

Aby naprawić implementację metody `callMethod`, należy skopiować zawartość obiektu `arguments`. Ponieważ potrzebne są tylko elementy po `obj` i `method`, do metody `slice` można przekazać początkowy indeks równy 2:

```
function callMethod(obj, method) {
  var args = [].slice.call(arguments, 2);
  return obj[method].apply(obj, args);
}
```

Teraz metoda `callMethod` działa zgodnie z oczekiwaniami:

```
var obj = {
  add: function(x, y) { return x + y; }
};
callMethod(obj, "add", 17, 25); // 42
```

Co warto zapamiętać?

- Nigdy nie modyfikuj obiektu `arguments`.
- Jeśli chcesz zmodyfikować zawartość obiektu `arguments`, najpierw skopiuj ją do zwykłej tablicy za pomocą instrukcji `[].slice.call(arguments)`.

Sposób 24. Używaj zmiennych do zapisywania referencji do obiektu `arguments`

Iterator to obiekt, który zapewnia sekwencyjny dostęp do kolekcji danych. Typowy interfejs API udostępnia metodę `next`, która zwraca następną wartość z sekwencji. Załóżmy, że chcesz napisać ułatwiającą pracę funkcję, która przyjmuje dowolną liczbę argumentów i tworzy iterator do poruszania się po tych wartościach:

```
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // 1
it.next(); // 4
it.next(); // 1
```

Funkcja `values` musi przyjmować dowolną liczbę argumentów, dlatego obiekt iteratora należy utworzyć tak, aby przechodził po elementach obiektu `arguments`:

```
function values() {
  var i = 0, n = arguments.length;
  return {
    hasNext: function() {
      return i < n;
    },
    next: function() {
      if (i >= n) {
        throw new Error("Koniec iteracji");
      }
      return arguments[i++]; // Nieprawidłowe argumenty
    }
  };
}
```

Jednak ten kod jest nieprawidłowy. Staje się to oczywiste przy próbie użycia iteratora:

```
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // undefined
```

```
it.next(); // undefined
it.next(); // undefined
```

Problem wynika z tego, że nowa zmienna `arguments` jest niejawnie wiązana w ciele każdej funkcji. Obiekt, który jest tu potrzebny, jest związany z funkcją `values`. Ale metoda `next` iteratora zawiera własną zmienną `arguments`. Dlatego gdy zwracana jest wartość `arguments[i++]`, pobierany jest argument z wywołania `it.next`, a nie jeden z argumentów funkcji `values`.

Rozwiązanie tego problemu jest proste — wystarczy związać nową zmienną lokalną w zasięgu potrzebnego obiektu `arguments` i zadbać o to, aby funkcje zagnieżdżone używały tylko tej jawnie nazwanej zmiennej:

```
function values() {
  var i = 0, n = arguments.length, a = arguments;
  return {
    hasNext: function() {
      return i < n;
    },
    next: function() {
      if(i >= n) {
        throw new Error("Koniec iteracji");
      }
      return a[i++];
    }
  };
}
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // 1
it.next(); // 4
it.next(); // 1
```

Co warto zapamiętać?

- Zwracaj uwagę na poziom zagnieżdżenia funkcji, gdy używasz obiektu `arguments`.
- Zwiąż z obiektem `arguments` referencję o jawnie określonym zasięgu, aby móc używać jej w funkcjach zagnieżdżonych.

Sposób 25. Używaj instrukcji bind do pobierania metod o stałym odbiorcy

Ponieważ nie istnieje różnica między metodą a właściwością, której wartością jest funkcja, łatwo można pobrać metodę obiektu i przekazać ją bezpośrednio jako wywołanie zwrotne do funkcji wyższego poziomu. Nie zapominaj jednak, że odbiorca pobranej funkcji nie jest na stałe ustawiony jako obiekt, z którego tę funkcję pobrano. Wyobraź sobie prosty obiekt bufora łańcuchów znaków, który zapisuje łańcuchy w tablicy, co umożliwi ich późniejsze scalenie:

```

var buffer = {
  entries: [],
  add: function(s) {
    this.entries.push(s);
  },
  concat: function() {
    return this.entries.join("");
  }
};

```

Wydaje się, że w celu skopiowania tablicy łańcuchów znaków do bufora można pobrać jego metodę `add` i wielokrotnie wywołać ją dla każdego elementu źródłowej tablicy, używając metody `forEach` ze standardu ES5:

```

var source = ["867", "-", "5309"];
source.forEach(buffer.add); // Błąd: elementy są niezdefiniowane

```

Jednak odbiorcą wywołania `buffer.add` nie jest obiekt `buffer`. Odbiorca funkcji zależy od sposobu jej wywołania, a nie jest ona wywoływana bezpośrednio w tym miejscu. Zamiast tego zostaje ona przekazana do metody `forEach`, której implementacja wywołuje funkcję w niedostępnym dla programisty miejscu. Okazuje się, że implementacja metody `forEach` jako domyślnego odbiorcy używa obiektu globalnego. Ponieważ obiekt globalny nie ma właściwości `entries`, przedstawiony kod zgłasza błąd. Na szczęście metoda `forEach` umożliwia podanie opcjonalnego argumentu, określającego odbiorcę wywołania zwrótnego. Dlatego można łatwo rozwiązać problem:

```

var source = ["867", "-", "5309"];
source.forEach(buffer.add, buffer);
buffer.join(); // "867-5309"

```

Nie wszystkie funkcje wyższego poziomu umożliwiają użytkownikom określenie odbiorcy wywołań zwrótnych. Jak rozwiązać problem, gdyby metoda `forEach` nie przyjmowała dodatkowego argumentu określającego odbiorcę? Dobrym rozwiązaniem jest utworzenie funkcji lokalnej, która wywołuje metodę `buffer.add` przy użyciu odpowiedniej składni:

```

var source = ["867", "-", "5309"];
source.forEach(function(s) {
  buffer.add(s);
});
buffer.join(); // "867-5309"

```

W tej wersji używana jest funkcja nakładkowa, która bezpośrednio wywołuje `add` jako metodę obiektu `buffer`. Zauważ, że sama funkcja nakładkowa w ogóle nie używa słowa kluczowego `this`. Niezależnie od sposobu wywołania tej funkcji nakładkowej (można ją wywołać jak zwykłą funkcję, jak metodę innego obiektu lub przy użyciu instrukcji `call`) zawsze przekazuje ona argument do docelowej tablicy.

Wersja funkcji wiążąca odbiorcę z konkretnym obiektem jest tworzona tak często, że w standardzie ES5 dodano obsługę tego wzorca w bibliotece. Obiekty

reprezentujące funkcje mają metodę `bind`, która przyjmuje obiekt odbiorcy i generuje funkcję nakładkową wywołującą pierwotną funkcję jako metodę odbiorcy. Za pomocą metody `bind` można uprościć przykładowy kod:

```
var source = ["867", "-", "5309"];
source.forEach(buffer.add.bind(buffer));
buffer.join(); // "867-5309"
```

Pamiętaj, że instrukcja `buffer.add.bind(buffer)` tworzy *nową* funkcję, zamiast modyfikować funkcję `buffer.add`. Nowa funkcja działa tak samo jak pierwotna, ale jej odbiorca to obiekt `buffer`. Pierwotna funkcja pozostaje niezmieniona. Oznacza to, że:

```
buffer.add === buffer.add.bind(buffer); // false
```

Jest to subtelna, ale ważna różnica. Oznacza to, że metodę `bind` można bezpiecznie wywołać nawet dla funkcji używanych w innych miejscach programu. Ma to znaczenie zwłaszcza w przypadku współużytkowanych metod z prototypów. Taka metoda będzie działać prawidłowo także dla obiektów potomnych prototypu. Więcej informacji o obiektach i prototypach znajdziesz w rozdziale 4.

Co warto zapamiętać?

- Pamiętaj, że pobranie metody nie prowadzi do ustawienia odbiorcy metody na obiekt, z którego ona pochodzi.
- Przy przekazywaniu metody obiektu do funkcji wyższego poziomu wykorzystaj funkcję anonimową, aby wywołać metodę dla odpowiedniego odbiorcy.
- Stosuj metodę `bind` do szybkiego tworzenia funkcji związanej z odpowiednim odbiorcą.

Sposób 26. Używaj metody bind do wiązania funkcji z podzbiorem argumentów (technika currying)

Metoda `bind` funkcji przydaje się nie tylko do wiązania metod z odbiorcami. Wyobraź sobie prostą funkcję tworzącą adresy URL na podstawie ich części składowych.

```
function simpleURL(protocol, domain, path) {
  return protocol + "://" + domain + "/" + path;
}
```

W programie potrzebne może być generowanie bezwzględnych adresów URL na podstawie ścieżek specyficznych dla witryny. Naturalnym sposobem na wykonanie tego zadania jest użycie metody `map` ze standardu ES5.

```
var urls = paths.map(function(path) {
    return simpleURL("http". siteDomain, path);
});
```

Zauważ, że w tej anonimowej funkcji w każdym powtórzeniu operacji przez metodę `map` używane są ten sam łańcuch znaków z protokołem i ten sam łańcuch znaków z domeną witryny. Dwa pierwsze argumenty funkcji `simpleURL` są niezmiennie w każdej iteracji. Potrzebny jest tylko trzeci argument. Można wykorzystać metodę `bind` funkcji `simpleURL`, aby automatycznie uzyskać potrzebną funkcję:

```
var urls = paths.map(simpleURL.bind(null, "http", siteDomain));
```

Wywołanie `simpleURL.bind` tworzy nową funkcję, która deleguje zadania do funkcji `simpleURL`. Jak zawsze w pierwszym argumencie metody `bind` podawany jest odbiorca. Ponieważ funkcja `simpleURL` go nie potrzebuje, można podać tu dowolną wartość (standardowo używane są wartości `null` i `undefined`). Argumenty przekazywane do funkcji `simpleURL` to wynik połączenia pozostałych argumentów funkcji `simpleURL.bind` z argumentami przekazanymi do nowej funkcji. Oznacza to, że gdy funkcja utworzona za pomocą instrukcji `simpleURL.bind` jest wywoływana z jednym argumentem `path`, w wyniku oddelegowania zadania wywołanie wygląda tak: `simpleURL("http", siteDomain, path)`.

Technika wiązania funkcji z podzbiorem argumentów to *currying* (nazwa pochodzi od logika Haskella Curry'ego, który spopularyzował tę metodę w matematyce). *Currying* pozwala na zwarte implementowanie delegowania i nie wymaga tak dużo szablonowego kodu jak jawne tworzenie funkcji nakładkowych.

Co warto zapamiętać?

- Za pomocą metody `bind` można utworzyć funkcję delegującą zadania, przekazującą stały podzbiór wymaganych argumentów. Ta technika to *currying*.
- Aby za pomocą tej techniki utworzyć funkcję, która ignoruje odbiorcę, jako reprezentujący go argument podaj wartość `null` lub `undefined`.

Sposób 27. Wybieraj domknięcia zamiast łańcuchów znaków do hermetyzowania kodu

Funkcje to wygodny sposób przechowywania kodu w postaci struktur danych i późniejszego uruchamiania go. Pozwala to na stosowanie zwięzłych abstrakcyjnych instrukcji wyższego poziomu, takich jak `map` i `forEach`, oraz jest istotą asynchronicznego wykonywania operacji wejścia-wyjścia w JavaScriptcie (zobacz rozdział 7.). Jednocześnie można też zapisać kod jako łańcuch znaków i przekazywać go do instrukcji `eval`. Programiści stoją więc przed wyborem, czy zapisać kod jako funkcję, czy jako łańcuch znaków.

Gdy masz wątpliwości, stosuj funkcje. Łańcuchy znaków zapewniają znacznie mniejszą swobodę. Wynika to z ważnego powodu — nie są domknięciami.

Przyjrzyj się prostej funkcji wielokrotnie powtarzającej określoną przez użytkownika operację:

```
function repeat(n, action) {
  for (var i = 0; i < n; i++) {
    eval(action);
  }
}
```

W zasięgu globalnym ta funkcja działa poprawnie, ponieważ wszystkie referencje do zmiennych występujące w łańcuchu znaków są interpretowane przez instrukcję `eval` jako zmienne globalne. Na przykład w skrypcie, który mierzy szybkość działania funkcji, można wykorzystać zmienne globalne `start` i `end` do przechowywania pomiarów czasu:

```
var start = [], end = [], timings = [];
repeat(1000,
  "start.push(Date.now()); f(); end.push(Date.now())");
for (var i = 0, n = start.length; i < n; i++) {
  timings[i] = end[i] - start[i];
}
```

Jednak ten skrypt jest podatny na problemy. Po przeniesieniu kodu do funkcji `start` i `end` nie będą już zmiennymi globalnymi:

```
function benchmark() {
  var start = [], end = [], timings = [];
  repeat(1000,
    "start.push(Date.now()); f(); end.push(Date.now())");
  for (var i = 0, n = start.length; i < n; i++) {
    timings[i] = end[i] - start[i];
  }
  return timings;
}
```

Ta funkcja powoduje, że instrukcja `repeat` wykorzystuje referencje do zmiennych globalnych `start` i `end`. W najlepszym przypadku jedna z tych zmiennych nie będzie istnieć, a wywołanie funkcji `benchmark` doprowadzi do błędu `ReferenceError`. Jeśli programista będzie miał pecha, kod wywoła instrukcję `push` dla globalnych obiektów związanych z nazwami `start` i `end`, a program będzie działał nieprzewidywalnie.

Bardziej odporny na błędy interfejs API przyjmuje funkcję zamiast łańcucha znaków:

```
function repeat(n, action) {
  for (var i = 0; i < n; i++) {
    action();
  }
}
```

Dzięki temu w skrypcie `benchmark` można bezpiecznie używać zmiennych lokalnych z domknięcia przekazywanego jako wielokrotnie uruchamiane wywołanie zwrotne:

```
function benchmark() {
  var start = [], end = [], timings = [];
  repeat(1000, function() {
    start.push(Date.now());
    f();
    end.push(Date.now());
  });
  for (var i = 0, n = start.length; i < n; i++) {
    timings[i] = end[i] - start[i];
  }
  return timings;
}
```

Inny problem z instrukcją `eval` polega na tym, że silniki o wysokiej wydajności mają zwykle większe problemy z optymalizacją kodu z łańcuchów znaków, ponieważ kod źródłowy może nie być dostępny dla kompilatora na tyle wcześnie, by można było w odpowiednim momencie przeprowadzić optymalizację. Wyrażenia funkcyjne można kompilować jednocześnie z kodem, w którym występują. Dlatego znacznie łatwiej się je kompiluje w standardowy sposób.

Co warto zapamiętać?

- Nigdy nie stosuj lokalnych referencji w łańcuchach znaków przekazywanych do interfejsu API, który wykonuje kod z łańcucha za pomocą instrukcji `eval`.
- Preferuj interfejsy API, które przyjmują wywoływane funkcje zamiast łańcuchów znaków przekazywanych do instrukcji `eval`.

Sposób 28. Unikaj stosowania metody `toString` funkcji

Funkcje w JavaScriptcie mają niezwykłą cechę — umożliwiają wyświetlenie swojego kodu źródłowego jako łańcucha znaków:

```
(function(x) {
  return x + 1;
}).toString(); // "function (x) {n    return x + 1;\n}"
```

Wyświetlanie kodu źródłowego funkcji za pomocą mechanizmu refleksji daje dużo możliwości, a pomysłowi hakerzy potrafią znaleźć ciekawe sposoby ich wykorzystania. Jednak metoda `toString` funkcji ma poważne ograniczenia.

Przed wszystkim standard ECMAScript nie określa żadnych wymagań wobec łańcuchów znaków zwracanych przez metodę `toString` funkcji. To oznacza, że różne silniki JavaScriptu mogą zwracać odmienne łańcuchy znaków. Możliwe nawet, że zwrócony tekst w rzeczywistości nie będzie podobny do kodu danej funkcji.

W praktyce silniki JavaScriptu *próbują* wyświetlić wierną reprezentację kodu źródłowego funkcji, o ile napisano ją w czystym JavaScriptcie. Nie sprawdza się to na przykład dla funkcji generowanych przez wbudowane biblioteki środowiska hosta:

```
(function(x) {  
    return x + 1;  
}).bind(16).toString(); // "function (x) {\n [native code]\n}"
```

Ponieważ w wielu środowiskach hosta funkcja `bind` jest zaimplementowana w innym języku programowania (zwykle w C++), zwracana jest skompilowana funkcja bez kodu źródłowego w JavaScriptcie, który środowisko mogłoby wyświetlić.

Ponieważ przeglądarki według standardu mogą w odpowiedzi na wywołanie funkcji `toString` zwracać inne dane, zbyt łatwo jest napisać program, który działa prawidłowo w jednym systemie, ale niepoprawnie w innym. Nawet drobne rozbieżności w implementacjach JavaScriptu (na przykład sposób formatowania odstępów) mogą zaburzyć działanie programu, który jest wrażliwy na szczegóły zapisu kodu źródłowego funkcji.

Ponadto kod źródłowy generowany przez metodę `toString` nie zwraca reprezentacji domknięcia z zachowaniem wartości związanych ze zmiennymi wewnętrznymi. Oto przykład:

```
(function(x) {  
    return function(y) {  
        return x + y;  
    }  
})(42).toString(); // "function (y) {\n return x + y;\n}"
```

Zauważ, że w wynikowym łańcuchu znaków występuje zmienna `x`, choć funkcja jest domknięciem wiążącym `x` z wartością `42`.

Te ograniczenia sprawiają, że trudno jest w przydatny i niezawodny sposób pobierać kod źródłowy funkcji. Dlatego zwykle warto unikać opisanej techniki. Do bardzo zaawansowanych operacji pobierania kodu źródłowego funkcji należy stosować starannie napisane parsery i biblioteki przetwarzające kod w JavaScriptcie. W razie wątpliwości najbezpieczniej jest traktować funkcje JavaScriptu jak abstrakcyjne struktury, których nie należy dzielić na fragmenty.

Co warto zapamiętać?

- Silniki JavaScriptu nie muszą wiernie zwracać kodu źródłowego funkcji po wywołaniu metody `toString`.
- Nigdy nie polegaj na szczegółach z pobranego kodu źródłowego funkcji, ponieważ wywołanie metody `toString` w różnych silnikach może dawać odmienne wyniki.

- Tekst zwracany przez metodę `toString` nie pokazuje wartości zmiennych lokalnych z domknięcia.
- Zwykle warto unikać wywoływania metody `toString` dla funkcji.

Sposób 29. Unikaj niestandardowych właściwości przeznaczonych do inspekcji stosu

Wiele środowisk JavaScriptu w przeszłości udostępniało mechanizmy do inspekcji **stosu wywołań**, czyli łańcucha obecnie wykonywanych aktywnych funkcji (więcej o stosie wywołań dowiesz się ze sposobu 64.). W starszych środowiskach hosta każdy obiekt `arguments` ma dwie dodatkowe właściwości: `arguments.callee` (określa funkcję wywołaną z argumentami `arguments`) i `arguments.caller` (określa funkcję wywołującą). Pierwsza z tych właściwości nadal jest obsługiwana w wielu środowiskach, jednak służy tylko do rekurencyjnego wskazywania funkcji anonimowych w nich samych.

```
var factorial = (function(n) {
    return (n <= 1) ? 1 : (n * arguments.callee(n - 1));
});
```

Nie jest to specjalnie przydatne, ponieważ łatwiej jest w funkcji wywołać ją za pomocą nazwy.

```
function factorial(n) {
    return (n <= 1) ? 1 : (n * factorial(n - 1));
}
```

Właściwość `arguments.caller` daje większe możliwości. Wskazuje funkcję, w której znalazło się wywołanie z danym obiektem `arguments`. Z powodów bezpieczeństwa mechanizm ten został usunięty z większości środowisk, tak więc może okazać się niedostępny. Wiele środowisk JavaScriptu udostępnia podobną właściwość dla obiektów funkcyjnych — niestandardową, ale często spotykaną właściwość `caller`. Określa ona jednostkę, która wywołała daną funkcję.

```
function revealCaller() {
    return revealCaller.caller;
}
```

```
function start() {
    return revealCaller();
}
```

```
start() === start; // true
```

Dobrym pomysłem może wydawać się wykorzystanie tej właściwości do pobierania *śladu stosu* (struktury danych zawierającej obecny stan stosu wywołań). Budowanie śladu stosu na pozór jest bardzo proste.

```
function getCallStack() {
    var stack = [];
```

```
    for (var f = getCallStack.caller; f; f = f.caller) {
        stack.push(f);
    }
    return stack;
}
```

Dla prostych stosów wywołań funkcja `getCallStack` działa poprawnie.

```
function f1() {
    return getCallStack();
}

function f2() {
    return f1();
}

var trace = f2();
trace; // [f1, f2]
```

Jednak działanie funkcji `getCallStack` łatwo jest zakłócić. Jeśli dana funkcja występuje w stosie wywołań więcej niż raz, kod odpowiedzialny za inspekcję stosu wpada w pętlę.

```
function f(n) {
    return n === 0 ? getCallStack() : f(n - 1);
}

var trace = f(1); // Pętla nieskończona
```

W czym tkwi problem? Ponieważ funkcja `f` rekurencyjnie wywołuje samą siebie, właściwość `caller` jest automatycznie ustawiana na `f`. Dlatego pętla w funkcji `getCallStack` nieustannie szuka funkcji `f`. Nawet jeśli programista spróbuje wykrywać takie cykle, niedostępne będą informacje o tym, jaka funkcja wywołała funkcję `f`, zanim funkcja `f` wywołała samą siebie. Informacje o reszcie stosu wywołań zostają więc utracone.

Wszystkie wymienione mechanizmy inspekcji stosu są niestandardowe oraz mają ograniczoną przenośność i zastosowania. Ponadto w standardzie ES5 są one niedozwolone w funkcjach w trybie *strict*. Próby dostępu do właściwości `caller` i `callee` funkcji w trybie *strict* oraz do obiektów `arguments` powodują błąd.

```
function f() {
    "use strict";
    return f.caller;
}

f(); // Błąd — nie można używać właściwości caller funkcji w trybie strict
```

Najlepsze podejście polega na rezygnacji z inspekcji stosu. Jeśli potrzebujesz sprawdzać stos na potrzeby debugowania, znacznie lepiej jest użyć interaktywnego debugera.

Co warto zapamiętać?

- Unikaj niestandardowych właściwości `arguments.caller` i `arguments.callee`, ponieważ w niektórych środowiskach są niedostępne.
- Unikaj niestandardowej właściwości `caller` funkcji, ponieważ nie zawsze zwraca kompletne informacje o stosie.

Skorowidz

A

- abstrakcje wyższego poziomu, 76
- aktor, 112
- API, 151
- argument, 81
- arność funkcji, 79
- ASCII, 43
- aspekty pragmatyczne, 13
- asynchroniczne
 - pętle, 190
 - wywołania zwrotne, 201
- atrapa, 167
- atrybut, 135
- automatyczne dodawanie średników, 37

B

- bezstanowy interfejs API, 161
- biblioteka, 151
 - Canvas, 161
- bitowe
 - operatory arytmetyczne, 28
 - wyrażenie OR, 26
- blok catch, 59
- blokowanie kolejki zdarzeń, 193
- błąd
 - parsowania, 38, 42
 - składni, 66
 - TypeError, 101
- błędy ignorowane, 187
- BMP, Basic Multilingual Plane, 44

C

- currying, 87, 88

D

- debugowanie, 63, 64
- dodawanie argumentów, 157
- domknięcia, 54, 88
- dziedziczenie
 - implementacji, 95
 - po klasach standardowych, 117

E

- ECMA, 13
- ECMAScript, 14
- enumeracja, 136

F

- format MediaWiki, 165
- funkcja, 71
 - Alert, 160
 - averageScore, 48
 - benchmark, 89
 - downloadAllAsync, 186, 188, 199
 - downloadAsync, 182, 184
 - downloadCachingAsync, 202
 - downloadOneAsync, 192
 - eval, 67–69
 - extend, 160
 - fail, 39
 - getCallStack, 93
 - isNaN, 29
 - make, 55
 - MEDIAWIKI, 165
 - negative, 192
 - next, 196
 - Number, 34
 - onsuccess, 198

funkcja

- sandwichMaker, 55
- score, 48
- select, 206
- showContents, 185
- simpleURL, 88
- status, 53
- trimSections, 58
- tryNextURL, 191
- User, 96, 102

funkcje

- asynchroniczne, 180
- blokujące, 180
- rekurencyjne, 193
- synchroniczne, 180
- wariadyczne, 79, 81
- wywoływane zwrrotnie, 74
- wyższego poziomu, 74

G

graf

- sceny, 111
- sieci społecznościowej, 137

H

- hermetyzowanie kodu, 88
- hierarchia dziedziczenia, 115
- hoisting, 57
 - zmiennych, 58

I

- idiom, 151
- IIFE, 23, 40, 61
- implementowanie słowników, 123
- instrukcja
 - __proto__, 95
 - apply, 79
 - bind, 85
 - break, 42
 - call, 77
 - continue, 42
 - eval, 90
 - getPrototypeOf, 95
 - if, 188
 - new, 101
 - Object.getPrototypeOf, 99
 - prototype, 95
 - return, 57
 - throw, 42

- var, 39
 - with, 52, 53
- interfejs API, 151
- interfejsy
 - bezstanowe, 161
 - stanowe, 161
 - elastyczne, 164
- introspekcja, 119
- iterator, 84

J

- jawna konwersja, 36
- jednostki kodowe, 43

K

- klasa, 98
 - Array, 118, 146
 - Dict, 124
 - MWPage, 165
 - User, 106
- kodowanie
 - o zmiennej długości, 45
 - znaków, 43
- koercja, 171
- kolejka zdarzeń, 179, 193
- kolekcja, 123
- kolekcje uporządkowane, 132
- konstrukcja try...catch, 59
- konstruktor, 71
 - klasy Array, 148
- kontekst, 112
- konwencje spójne, 151
- konwersja typu, 28
 - jawna, 36
 - niejawna, 35

L

- liczba argumentów, 79
- liczby
 - o podwójnej precyzji, 26
 - zmiennoprzecinkowe, 24
- literały tablicowe, 148

Ł

- łańcuch
 - zasięgu, 52
 - znaków, 30, 32, 43–45
 - metod, 174

łączenie
 metod w łańcuch, 175
 obietnic, 204
 plików, 23
 łączność lewostronna, 28

M

mapy deskryptorów właściwości, 126
 mechanizm naprawiania błędów, 38
 metadata, 135
 metoda, 71
 bind, 87
 call, 77, 82
 concat, 147
 enable, 168, 172
 forEach, 78, 146
 hasOwnProperty, 128
 inNetwork, 195
 Object.create, 102, 114
 pick, 139
 postMessage, 194
 replace, 174
 shift, 82
 slice, 83, 148
 toString, 25, 30, 90
 toUpperCase, 33
 valueOf, 30, 31, 34
 metody
 dla niestandardowego odbiorcy, 77
 do obsługi iteracji, 142
 klasy Array, 146, 148
 niedeterministyczne, 139
 o stałym odbiorcy, 85
 w prototypach, 103
 modyfikowanie
 obiektu, 82, 136
 właściwości `__proto__`, 100
 monkey patching, 120

N

nakładki obiektowe, 32
 naprawianie błędów, 38
 narzędzie lint, 51
 nazwa właściwości, 115
 niejawną konwersja typu, 27
 niejawne
 przenoszenie deklaracji zmiennych,
 57
 tworzenie nakładek obiektowych, 34
 wiązanie obiektu, 109

nieprzenośne określanie zasięgu, 62, 65
 niestandardowy odbiorca, 77

O

obiekt, 95
 arguments, 82, 84
 this, 109, 177
 typu String, 32
 XMLHttpRequest, 182
 obiekty
 globalne, 48
 z opcjami, 157
 obietnice, 203
 obsługa
 błędów, 187
 iteracji, 142
 łańcuchów metod, 174
 ograniczone konstrukcje, 41
 określanie typu na podstawie struktury,
 166
 operacja łączna lewostronnie, 28
 operator
 +, 30
 ++, 42
 identyczności, 34
 równości, 33, 35
 typeof, 32
 undefined, 32
 operatory
 arytmetyczne bitowe, 28
 przesunięcia, 28
 osadzany język skryptowy, 179

P

parametr formalny, 81
 pary surogatów, 46
 pętla
 for, 60, 140, 142
 for...in, 134, 135, 140
 while, 140
 zdarzeń, 179, 181
 pliki o różnych trybach, 23
 pobieranie
 metod, 85
 prototypów obiektów, 97
 podstawa, 25
 podwójna precyzja, 25
 praca na odległość, 120
 prawo łączności, 26
 predykat, 143

programowanie
 asynchroniczne, 187
 defensywne, 172
 obiektowe, 71
 prototyp, 95, 119
 Array.prototype, 126
 Object.prototype, 134
 prototypy null, 126
 przechowywanie
 metod
 w egzemplarzach, 104
 w prototypie, 105
 prywatnych danych, 105
 stanu egzemplarza, 107
 w egzemplarzach, 108
 w prototypie, 108
 przeciążony operator, 30
 przekazywanie argumentów, 157
 przepelnienie stosu, 192
 przesłanianie konstruktora, 103
 przestrzeń nazw, 48
 przetwarzanie skrócone, 145

R

referencje do obiektu arguments, 84
 rekurencja, 190

S

scalanie skryptów, 22, 24
 semantyka, 13
 składnia, 13
 słowniki, 123
 słowo kluczowe
 arguments, 81
 const, 20
 new, 101
 return, 41
 this, 72, 86
 var, 64
 with, 51
 standard
 ECMAScript, 13, 19
 IEEE, 29
 stanowe interfejsy API, 162
 stos wywołań, 92, 191, 193
 stosowanie
 domknięć, 105
 enumerowanych właściwości, 134
 instrukcji Object.getPrototypeOf, 99
 metody toString, 90

nazwanych wyrażen funkcyjnych, 62
 rekurencji, 190
 tablic, 132
 techniki monkey patching, 120
 struktura, 164
 styl płynny, 176
 systemy modularne, 23

Ś

śląd stosu, 93
 średnik, 37
 środowisko leksykalne, 52

T

tablice, 123
 technika monkey patching, 120
 tryb strict, 21, 23
 tworzenie
 abstrakcji, 76
 asynchronicznych pętli, 190
 funkcji wariadycznych, 81
 kolekcji, 132
 pętli, 145
 zmiennych globalnych, 47
 zmiennych lokalnych, 48
 typy
 oparte na strukturze, 169
 proste, 32

U

UCS-2, 43
 Unicode, 43
 UTF-16, 45

W

wartości
 logiczne, 31
 oznaczające fałsz, 31
 wartość
 NaN, 29
 null, 28, 63
 ostateczna, 204
 scores.length, 141
 true, 31
 undefined, 32, 153, 159
 wątek, 180
 wektor bitowy, 167
 wersja JavaScriptu, 19

- wiązanie funkcji, 87, 88
- właściwości
 - enumerowane, 134
 - wewnętrzne, 117
- właściwość
 - __proto__, 97, 100, 127, 130
 - caller, 94
 - info, 53
 - length, 168
 - prototype, 96
- współbieżne pobieranie plików, 197
- współbieżność, 14
 - oparta na pętli zdarzeń, 179
- współrzędna kodowa znaku, 43
- współrzędne kodowe Unicode, 46
- wykonywanie funkcji rekurencyjnej, 193
- wypełniacz, 121
- wyrażenia
 - funkcyjne, 23, 56
 - anonimowe, 64
 - natychmiast wywoływane, 61
 - nazwane, 62
 - IIFE, 59
- wyrażenie OR, 25
- wyścig do danych, 199
- wywołania
 - asynchroniczne, 183
 - bezpośrednie, 70
 - funkcji, 71, 79
 - konstruktorów, 71
 - metod, 71, 77
 - pośrednie, 70
 - zwrotne, 183
 - nazwane, 183
 - zagnieżdżone, 183
- wywołanie downloadAsync, 183

Z

- zagnieżdżanie, 184
 - deklaracji funkcji, 65
- zasięg
 - blokowy, 57
 - leksykalny, 57
 - lokalny, 59
 - zmiennych, 47
- zaśmiecianie przez prototypy, 125, 126, 128
- zbiór łańcuchów znaków, 168
- zdarzenia, 180
- zmienna arguments, 21
- zmiennie
 - globalne, 23, 47
 - lokalne, 50, 67
- znak
 - (, 38
 - ., 46
 - /, 39
 - [, 38
 - ;, 41
- znaki niebezpieczne, 40
- zwracanie obiektu this, 177

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**

Najlepsze porady dotyczące JavaScriptu dla każdego programisty!

JavaScript jeszcze do niedawna kojarzył się głównie ze stronami oraz z aplikacjami internetowymi, a jego głównym zastosowaniem były operacje na drzewie DOM. Jednak te czasy mijają, a język ten jest coraz chętniej wykorzystywany również po stronie serwera. JavaScript jako pełnoprawny język programowania? Oczywiście! W dodatku okazuje się, że może on być bardzo wydajny, elastyczny i przyjazny dla programistów — wystarczy przestrzegać kilku zasad!

Te tajemnicze zasady zostały zebrane w niniejszej książce. Jeśli będziesz o nich pamiętać, wykorzystasz w pełni potencjał JavaScriptu. W trakcie lektury dowiesz się, jak najlepiej deklarować zmienne, używać funkcji oraz radzić sobie z obiektami i prototypami. W kolejnych rozdziałach nauczysz się budować przyjazne API oraz korzystać ze słowników z tablic. Na sam koniec zdobędziesz informacje, które mają kluczowe znaczenie w przypadku programowania współbieżnego. Jeżeli jesteś programistą JavaScript, jeżeli chcesz poprawić swoje umiejętności programowania w tym języku, jest to dla Ciebie lektura obowiązkowa. Przekonaj się, jak przyjemne i wydajne może być programowanie w JavaScriptcie!

Dzięki tej książce:

- nauczysz się minimalizacji liczby obiektów globalnych
- opanujesz stosowanie wyrażen IIFE do tworzenia zasięgu lokalnego
- dowiesz się, jak nie blokować kolejki zdarzeń operacjami wejścia-wyjścia
- poznasz różnice pomiędzy instrukcjami prototype, getPrototypeOf i __proto__

David Herman — starszy specjalista w firmie Mozilla Research, członek grupy ECMA TC39, odpowiedzialnej za rozwój standardu języka JavaScript. Ma wkład w liczne projekty, takie jak: task.js, sweet.js, asm.js, Rust.

 **Addison-Wesley**
Pearson Education

	
37559	numer katalogowy
księgarnia internetowa	
http://helion.pl	
zamówienia telefoniczne	
	0 801 339900
	0 601 339900
Informatyka w najlepszym wydaniu	

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

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

