

ZROZUMIEĆ JAVASCRIPT

Wprowadzenie do programowania

Marijn Haverbeke



Helion 



Tytuł oryginału: Eloquent JavaScript, Second Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0969-2

Copyright © 2015 by Marijn Haverbeke. Title of English-language original: Eloquent JavaScript, 2nd Edition, ISBN 978-1-59327-584-6, published by No Starch Press.

Polish language edition copyright © 2015 by Helion SA.
All rights reserved.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/zrojsc.zip>

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

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

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

Spis treści

WPROWADZENIE	19
O programowaniu	20
Dlaczego język jest ważny	22
Co to jest JavaScript	24
Kod... — co to właściwie jest	25
Książka w ogólnym zarysie	26
Konwencje typograficzne	27

CZĘŚĆ I. JĘZYK

29

I

WARTOŚCI, TYPY I OPERATORY	31
Wartości	32
Liczby	32
Arytmetyka	34
Liczby specjalne	34
Łańcuchy	35
Operatory jednoargumentowe	36
Wartości logiczne	37
Porównywanie	37
Operatory logiczne	38
Wartości niezdefiniowane	39
Automatyczna konwersja typów	39
Skrócona metoda wyznaczania wartości wyrażeń logicznych	40
Podsumowanie	41

2

STRUKTURA PROGRAMU	43
Wyrażenia i instrukcje	43
Zmienne	44
Słowa kluczowe i zarezerwowane	46
Środowisko	46
Funkcje	47
Funkcją console.log	47
Wartości zwrotne	48
Funkcje prompt i confirm	48
Sterowanie sposobem wykonywania programu	49
Wykonywanie warunkowe	50
Pętle while i do	51
Stosowanie wcięć w kodzie	53
Pętla for	54
Wychodzenie z pętli	55
Zwięzłe modyfikowanie zmiennych	55
Rozdzielanie zadań przy użyciu instrukcji switch	56
Stosowanie wielkich liter	57
Komentarze	57
Podsumowanie	58
Ćwiczenia	59
Pętlowy trójkąt	59
FizzBuzz	59
Plansza do gry w szachy	59

3

FUNKCJE	63
Definiowanie funkcji	64
Parametry i zakresy dostępności	65
Zagnieżdżone zakresy dostępności	66
Funkcje jako wartości	67
Sposób deklarowania funkcji	67
Stos wywołań	68
Argumenty opcjonalne	70
Zamknięcia	71
Rekurencja	72
Hodowanie funkcji	75
Funkcje i skutki uboczne	77
Podsumowanie	78
Ćwiczenia	78
Minimum	78
Rekurencja	78
Liczenie znaków	79

4

STRUKTURY DANYCH — OBIEKTY I TABLICE	81
Wiewiórkołak	82
Zbiory danych	82
Własności	83
Metody	84
Obiekty	85
Zmiennosc	88
Dziennik wiewiórkołaka	89
Obliczanie korelacji	90
Obiekty jako słowniki	92
Ostateczna analiza	93
Dalsza tablicologia	95
Łącuchy i ich własności	96
Obiekt arguments	97
Obiekt Math	98
Obiekt globalny	100
Podsumowanie	100
Ćwiczenia	101
Suma przedziału liczb	101
Odwracanie tablicy	101
Lista	102
Porównywanie głębokie	102

5

FUNKCJE WYŻSZEGO RZĘDU	105
Abstrakcja	106
Abstrakcja operacji przeglądania tablicy	107
Funkcje wyższego rzędu	109
Przekazywanie argumentów	110
JSON	111
Filtrowanie tablicy	112
Przekształcanie tablic za pomocą metody map	113
Podsumowywanie przy użyciu metody reduce	114
Składalność	115
Koszty	116
Prapraprapra...	116
Wiązanie	119
Podsumowanie	120
Ćwiczenia	120
Spłaszczanie	120
Różnica wieku między matką i dzieckiem	120
Historyczna średnia długość życia	120
Wszystko i trochę	121

6

SEKRETNE ŻYCIE OBIEKTÓW	123
Historia	123
Metody	125
Prototypy	126
Konstruktory	127
Przesłanianie dziedziczonych własności	128
Interferencja prototypów	129
Obiekty bez prototypów	131
Polimorfizm	132
Formowanie tabeli	132
Metody pobierające i ustawiające	137
Dziedziczenie	139
Operator instanceof	140
Podsumowanie	141
Ćwiczenia	141
Typ wektorowy	141
Kolejna komórka	142
Interfejs sekwencyjny	142

7

PROJEKT — ELEKTRONICZNE ŻYCIE	145
Definicja	145
Reprezentacja przestrzeni	146
Interfejs programistyczny stworzeń	148
Obiekt World	149
Zmienna this i jej zakres dostępności	151
Animacja życia	153
Rusza się	155
Więcej form życia	156
Bardziej realistyczna symulacja	157
Funkcje obsługi czynności	158
Populacja nowego świata	160
Ożywianie świata	161
Ćwiczenia	162
Sztuczna głupota	162
Drapieżniki	163

8

BŁĘDY I OBSŁUGA BŁĘDÓW	165
Błędy programisty	165
Tryb ścisły	166
Testowanie	167
Debugowanie	168
Propagacja błędów	170
Wyjątki	171

Sprzątanie po wyjątkach	172
Selektywne przechwytywanie wyjątków	174
Asercje	176
Podsumowanie	177
Ćwiczenia	177
Spróbuj jeszcze raz	177
Zamknięte pudełko	177

9

WYRAŻENIA REGULARNE 181

Tworzenie wyrażeń regularnych	182
Dopasowywanie wzorców	182
Dopasowywanie zbiorów znaków	183
Powtarzanie części wzorca	184
Grupowanie podwyrażeń	185
Dopasowania i grupy	186
Typ Date	187
Granice słów i łańcuchów	188
Wzorce wyboru	189
Zasady dopasowywania	189
Wycofywanie	190
Metoda replace	192
Zachłanność	193
Dynamiczne tworzenie obiektów RegExp	195
Metoda search	195
Własność lastIndex	196
Przeglądanie dopasowanych elementów za pomocą pętli	197
Przetwarzanie plików INI	198
Znaki międzynarodowe	200
Podsumowanie	200
Ćwiczenia	201
Wyrażeniowy golf	201
Rodzaje cudzysłowów	202
Jeszcze raz liczby	202

10

MODUŁY 203

Co dają moduły	203
Przestrzenie nazw	204
Wielokrotne wykorzystywanie kodu	204
Rozluźnienie powiązań	205
Funkcje w roli przestrzeni nazw	206
Obiekty jako interfejsy	207
Pomijanie zakresu globalnego	208
Wykonywanie danych jako kodu	209

Funkcja dołączająca	209
Powolne wczytywanie modułów	211
Projektowanie interfejsu	214
Przewidywalność	214
Możliwość składania	215
Interfejsy warstwowe	215
Podsumowanie	216
Ćwiczenia	216
Nazwy miesięcy	216
Powrót do elektronicznego życia	216
Zależności cykliczne	217

II

PROJEKT — JĘZYK PROGRAMOWANIA 219

Analiza składni	219
Ewaluator	223
Specjalne konstrukcje	224
Środowisko	226
Funkcja	228
Kompilacja	229
Ściąganie	229
Ćwiczenia	230
Tablice	230
Zamknięcie	230
Komentarze	231
Naprawienie zakresu	231

CZĘŚĆ II. PRZEGLĄDARKI INTERNETOWE 233

12

JAVASCRIPT I PRZEGLĄDARKI INTERNETOWE 235

Sieci i internet	236
Sieć ogólnoswiatowa	237
HTML	238
HTML i JavaScript	240
Piaskownica	241
Zgodność i wojny przeglądarek	241

13

OBIEKTOWY MODEL DOKUMENTU 243

Struktura dokumentu	243
Drzewa	245
Standard	246
Poruszanie się po drzewie	246
Znajdowanie elementów	248

Modyfikowanie dokumentu	249
Tworzenie węzłów	249
Atrybuty	251
Rozmieszczenie elementów na stronie	253
Style	255
Kaskadowe arkusze stylów	257
Selektory	258
Pozycjonowanie i animowanie	259
Podsumowanie	261
Ćwiczenia	262
Budowa tabeli	262
Elementy według nazwy znacznika	262
Kapelusz kota	263

14

OBSŁUGA ZDARZEŃ	265
Procedury obsługi zdarzeń	265
Zdarzenia i węzły DOM	266
Obiekty zdarzeń	267
Propagacja	268
Działania domyślne	269
Zdarzenia klawiszy	270
Kliknięcia myszą	272
Ruch myszy	273
Zdarzenia przewijania	275
Zdarzenia aktywacji	276
Zdarzenie load	277
Czas wykonywania skryptu	277
Zegary	279
Eliminowanie skutków zbyt częstego wyzwalania zdarzeń	280
Podsumowanie	281
Ćwiczenia	282
Cenzura klawiatury	282
Trop myszy	282
Karty	282

15

PROJEKT — GRA PLATFORMOWA	285
Gra	286
Technologia	286
Poziomy	287
Wczytywanie poziomu	288
Aktorzy	289
Hermetyzacja jako obciążenie	291
Rysowanie	292

Ruch i kolizje	297
Aktorzy i czynności	299
Śledzenie klawiszy	303
Uruchamianie gry	303
Ćwiczenia	306
Koniec gry	306
Wstrzymywanie gry	306

16

RYSOWANIE NA KANWIE	309
SVG	310
Kanwa	311
Wypełnienie i obrys	312
Ścieżki	312
Krzywe	314
Rysowanie wykresu kołowego	317
Tekst	318
Obrazy	319
Przekształcenia	320
Zapisywanie i kasowanie przekształceń	323
Powrót do gry	324
Wybór interfejsu graficznego	329
Podsumowanie	330
Ćwiczenia	331
Kształty	331
Wykres kołowy	331
Odbijająca się piłka	332
Obliczenia na zapas	332

17

HTTP	335
Protokół	335
Przeglądarki i HTTP	337
XMLHttpRequest	339
Wysyłanie żądania	339
Żądania asynchroniczne	341
Pobieranie danych XML	341
Piaskownica dla HTTP	342
Abstrahowanie żądań	343
Obietnice	345
Docenianie HTTP	348
Bezpieczeństwo i HTTPS	348
Podsumowanie	349
Ćwiczenia	350
Negocjacja treści	350
Oczekiwanie na wiele obietnic	350

18

FORMULARZE I POLA FORMULARZA	353
Pola	354
Aktywacja	355
Wyłączanie pól	356
Formularz jako całość	357
Pola tekstowe	358
Pola wyboru i przyciski radiowe	359
Pola opcji do wyboru	360
Pola plikowe	362
Zapisywanie danych u klienta	364
Podsumowanie	366
Ćwiczenia	367
Pracownia JavaScript	367
Automatyczne uzupełnianie	367
Gra w życie Conwaya	367

19

PROJEKT — PROGRAM RYSUNKOWY	369
Implementacja	370
Tworzenie modelu DOM	370
Podstawa	371
Wybór narzędzi	372
Kolor i rozmiar pędzla	374
Zapisywanie	376
Wczytywanie obrazów z plików	377
Wykończenie	379
Ćwiczenia	380
Prostokąty	380
Próbnik kolorów	381
Wypełnianie zalewowe	382

CZĘŚĆ III. WIĘCEJ NIŻ JAVASCRIPT

385

20

NODE.JS	387
Podstawy	388
Asynchroniczność	388
Polecenie node	389
Moduły	391
Instalowanie modułów z repozytorium NPM	392
Moduł systemu plików	393
Moduł HTTP	395
Strumienie	396

Prosty serwer plików	398
Obsługa błędów	402
Podsumowanie	404
Ćwiczenia	404
Negocjacja treści raz jeszcze	404
Tamowanie wycieku	405
Tworzenie katalogów	405
Publiczna przestrzeń w internecie	406

21

PROJEKT — SERWIS DLA PASJONATÓW 407

Projekt	408
Długie sondowanie	409
Interfejs HTTP	410
Serwer	412
Trasowanie	412
Serwowanie plików	413
Przemowy jako zasoby	414
Długie sondowanie	416
Klient	419
HTML	419
Uruchamianie	421
Wyświetlanie przemów	422
Aktualizowanie serwera	424
Pokazywanie zmian	426
Ćwiczenia	427
Zapisywanie danych na dysku	427
Resetowanie pól komentarzy	427
Lepsze szablony	427
Wykluczeni ze skryptów	428

22

WYDAJNOŚĆ JAVASCRIPTU 429

Kompilacja etapowa	430
Układ grafowy	431
Definiowanie grafu	432
Pierwsza funkcja rozkładu ukierunkowanego siłowo	433
Profilowanie	435
Rozwijanie funkcji	437
Powrót do klasycznych pętli	438
Unikanie pracy	438
Zmniejszanie ilości śmieci	439
Usuwanie nieużytków	440
Zapisywanie danych w obiektach	441
Typy dynamiczne	443

Podsumowanie	444
Ćwiczenia	444
Znajdowanie drogi	444
Mierzenie czasu	445
Optymalizacja	446
PODPowiedzi do Ćwiczeń	447
Struktura programu	447
Pętlowy trójkąt	447
FizzBuzz	448
Plansza do gry w szachy	448
Funkcje	448
Minimum	448
Rekurencja	448
Liczenie znaków	449
Struktury danych — obiekty i tablice	449
Suma przedziału liczb	449
Odwracanie tablicy	449
Lista	450
Porównywanie głębokie	450
Funkcje wyższego rzędu	451
Różnica wieku między matką i dzieckiem	451
Historyczna średnia długość życia	451
Wszystko i trochę	451
Sekretne życie obiektów	452
Typ wektorowy	452
Kolejna komórka	452
Interfejs sekwencyjny	452
Projekt — elektroniczne życie	453
Sztuczna głupota	453
Drapieżniki	453
Błędy i obsługa błędów	453
Spróbuj jeszcze raz	453
Zamknięte pudełko	454
Wyrażenia regularne	454
Rodzaje cudzysłowów	454
Jeszcze raz liczby	454
Moduły	455
Nazwy miesięcy	455
Powrót do elektronicznego życia	455
Zależności cykliczne	456
Projekt — język programowania	456
Tablice	456
Zamknięcie	456
Komentarze	456
Naprawienie zakresu	457

Obiektowy model dokumentu	457
Budowa tabeli	457
Elementy według nazwy znacznika	457
Obsługa zdarzeń	458
Cenzura klawiatury	458
Trop myszy	458
Karty	458
Projekt — gra platformowa	459
Koniec gry	459
Wstrzymywanie gry	459
Rysowanie na kanwie	460
Kształty	460
Wykres kołowy	460
Odbijająca się piłka	461
Obliczenia na zapas	461
HTTP	461
Negocjacja treści	461
Oczekiwanie na wiele obietnic	462
Formularze i pola formularza	462
Pracownia JavaScript	462
Automatyczne uzupełnianie	462
Gra w życie Conwaya	463
Projekt — program rysunkowy	463
Prostokąty	463
Próbnik kolorów	464
Wypełnianie zalewowe	464
Node.js	465
Negocjacja treści raz jeszcze	465
Tamowanie wycieku	465
Tworzenie katalogów	466
Publiczna przestrzeń w internecie	466
Projekt — serwis dla pasjonatów	466
Zapisywanie danych na dysku	466
Resetowanie pól komentarzy	467
Lepsze szablony	467
Wykluczeni ze skryptów	467
Wydajność JavaScriptu	468
Znajdowanie drogi	468
Optymalizacja	468

SKOROWIDZ **469**

5

Funkcje wyższego rzędu

Duży program to drogi program i to nie tylko z powodu ilości czasu, jaką trzeba poświęcić na jego napisanie. Rozmiar prawie zawsze wiąże się z poziomem złożoności, a ta z kolei mąci w głowie programistom. Natomiast zdezorientowany programista może popełniać błędy. Ponadto w dużym programie jest wiele miejsc do ukrycia błędów.

Wróćmy na chwilę do ostatnich dwóch przykładów z wprowadzenia. Pierwszy jest samodzielny i zawiera sześć wierszy kodu źródłowego.

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

Drugi wykorzystuje dwie funkcje zewnętrzne i zajmuje tylko jedną linijkę.

```
console.log(sum(range(1, 10)));
```

W którym jest większe ryzyko popełnienia błędu?

Jeśli weźmiemy pod uwagę rozmiary definicji funkcji `sum` i `range`, to drugi program też jest duży — nawet większy od pierwszego. Ale i tak jestem gotów się spierać, że w tym programie jest mniejsze ryzyko wystąpienia błędu.

Swoje twierdzenie opieram na tym, że rozwiązanie w tym programie jest przedstawione za pomocą słów, które dość dokładnie opisują problem. Sumowanie przedziałów liczb nie polega na używaniu pętli i liczników, tylko na wykorzystaniu przedziałów liczbowych i sum.

Definicje tych słów (funkcji `sum` i `range`) oczywiście muszą zawierać pętle, liczniki i inne szczegóły implementacyjne. Ale jako że wyrażają prostsze pojęcia niż program jako całość, łatwiej jest napisać je bez błędów.

Abstrakcja

W programowaniu tego rodzaju słowniki nazywa się **abstrakcjami**. Ukrywają one szczegóły i umożliwiają programiście wyrażanie problemów na nieco wyższym (bardziej abstrakcyjnym) poziomie.

Porównaj dwa poniższe przepisy na zupełną grochową:

Wsyp szklankę suszonego grochu na osobę do pojemnika. Dodaj tyle wody, aby przykryć groch. Odstaw pojemnik na przynajmniej 12 godzin. Wyjmij groch z wody i przelóż go do garnka. Wlej cztery szklanki wody na osobę. Przykryj garnek i gotuj na wolnym ogniu dwie godziny. Obierz pół cebuli na osobę. Pokrój cebulę nożem na kawałki i dodaj ją do garnka. Weź po jednym selerze na osobę i pokrój go nożem, a następnie też dodaj do garnka. Weź po jednej marchewce na osobę. Pokrój ją na kawałki. Użyj noża! Wrzuć warzywo do garnka. Gotuj jeszcze dziesięć minut.

I drugi przepis:

Dla każdej osoby: jedna szklanka suszonego luskanego grochu, pół pokrojonej cebuli, seler i marchewka.

Namoczyć groch przez 12 godzin, gotować na wolnym ogniu w czterech szklankach wody (na osobę). Pokroić i dodać warzywa. Gotować jeszcze 10 minut.

Drugi przepis jest krótszy i czytelniejszy. Ale trzeba rozumieć parę słów odnoszących się do gotowania — *namoczyć*, *gotować na wolnym ogniu*, *pokroić* i może *warzywo*.

Programista jednak nie może oczekiwać, że wszystkie potrzebne mu słowa będą czekać na niego w słowniku. Przez to może dać się wciągnąć w postępowanie zilustrowane przez pierwszy przykład — dokładny opis wszystkich kroków do wykonania przez komputer bez zwracania uwagi na pojęcia wyższego rzędu, które wyrażają poszczególne słowa.

Programista z doświadczeniem powinien nabyć umiejętność rozpoznawania sytuacji, w których aż się prosi, żeby wyabstrahować jakiś koncept w postaci nowego słowa.

Abstrakcja operacji przeglądania tablicy

Zwykle funkcje, takie jak oglądane do tej pory, są bardzo dobrym narzędziem abstrakcji. Ale czasami ich możliwości okazują się niewystarczające.

W poprzednim rozdziale kilka razy wystąpiła pętla for tego typu:

```
var array = [1, 2, 3];
for (var i = 0; i < array.length; i++) {
  var current = array[i];
  console.log(current);
}
```

Pętlę tę można przeczytać tak: „Dla każdego elementu w tablicy dokonaj zapisu w konsoli”. Ale jest to dość zawily sposób działania polegający na użyciu zmiennej licznikowej *i*, testu porównawczego z długością tablicy oraz dodatkowej deklaracji zmiennej do pobrania bieżącego elementu. Oprócz zwykłej brzydoty kod ten ma jeszcze tę cechę, że łatwo w nim się pomylić. Można przypadkowo po raz kolejny użyć zmiennej *i*, źle napisać nazwę `length` jako `lenght`, pomylić zmienną *i* z `current` itd.

W świetle tych faktów można zdecydować się na napisanie funkcji.

Wiesz, jak to zrobić?

W sumie nie ma nic trudnego w napisaniu funkcji przeglądającej tablicę i wywołującej funkcję `console.log` na każdym elemencie tej struktury.

```
function logEach(array) {
  for (var i = 0; i < array.length; i++)
    console.log(array[i]);
}
```

A co, jeśli zechcemy każdy element poddać innej operacji? Jako że „robienie czegoś” reprezentuje się jako funkcję, a funkcje to wartości, możemy przekazać naszą czynność jako wartość funkcyjną.

```
function forEach(array, action) {
  for (var i = 0; i < array.length; i++)
    action(array[i]);
}

forEach(["Wampeter", "Foma", "Granfalloon"], console.log);
// → Wampeter
// → Foma
// → Granfalloon
```

Zamiast przekazywać gotową funkcję do `forEach`, można utworzyć wartość funkcyjną na miejscu.

```
var numbers = [1, 2, 3, 4, 5], sum = 0;
forEach(numbers, function(number) {
    sum += number;
});
console.log(sum);
// → 15
```

To wygląda jak klasyczna pętla `for` z treścią główną wpisaną jako blok pod spodem. Ale teraz treść ta znajduje się w wartości funkcyjnej, jak również w nawiasie wywołania funkcji `forEach`. Dlatego właśnie potrzebne jest zamknięcie klamry i nawiasu.

Stosując tę technikę, można określić nazwę zmiennej dla bieżącego elementu (`number`), zamiast wybierać ją ręcznie z tablicy.

Tak naprawdę to nie musimy pisać funkcji `forEach`, ponieważ jest to standardowa metoda tablic. A ponieważ tablica jest już dostarczona jako obiekt, na którym działa metoda, do `forEach` przekazuje się tylko jeden wymagany argument — funkcję do wykonania na każdym elemencie.

Aby pokazać, jak bardzo jest to przydatne, wróć do funkcji z poprzedniego rozdziału. Zawiera ona dwie pętle przeglądające tablice.

```
function gatherCorrelations(journal) {
    var phis = {};
    for (var entry = 0; entry < journal.length; entry++) {
        var events = journal[entry].events;
        for (var i = 0; i < events.length; i++) {
            var event = events[i];
            if (!(event in phis))
                phis[event] = phi(tableFor(event, journal));
        }
    }
    return phis;
}
```

Przy użyciu funkcji `forEach` można ten kod trochę skrócić i uprościć.

```
function gatherCorrelations(journal) {
    var phis = {};
    journal.forEach(function(entry) {
        entry.events.forEach(function(event) {
            if (!(event in phis))
                phis[event] = phi(tableFor(event, journal));
        });
    });
    return phis;
}
```

Funkcje wyższego rzędu

Funkcje działające na innych funkcjach, pobierające je jako argument lub je zwracające, nazywają się **funkcjami wyższego rzędu**. Jeśli już przyjmujesz do wiadomości fakt, że funkcje to zwykłe wartości, to istnienie takich funkcji nie wyda Ci się ani trochę dziwne. Określenie to pochodzi z matematyki, w której bardziej precyzyjnie odróżnia się funkcje od innych wartości.

Funkcje wyższego rzędu umożliwiają abstrahowanie *czynności*, nie wartości. Występują w paru postaciach. Na przykład można utworzyć funkcję tworzącą nowe funkcje.

```
function greaterThan(n) {
  return function(m) { return m > n; };
}
var greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

Można też tworzyć funkcje zmieniające inne funkcje.

```
function noisy(f) {
  return function(arg) {
    console.log("wywoływanie z ", arg);
    var val = f(arg);
    console.log("wywołano z ", arg, "- otrzymano", val);
    return val;
  };
}
noisy(Boolean)(0);
// → wywoływanie z 0
// → wywołano z 0 - otrzymano false
```

Można nawet pisać funkcje implementujące nowe typy kontroli przepływu sterowania.

```
function unless(test, then) {
  if (!test) then();
}
function repeat(times, body) {
  for (var i = 0; i < times; i++) body(i);
}

repeat(3, function(n) {
  unless(n % 2, function() {
    console.log(n, "to liczba parzysta");
  });
});
```

```
});  
// → 0 to liczba parzysta  
// → 2 to liczba parzysta
```

Opisany w rozdziale 3. zakres leksykalny działa w tym przypadku na naszą korzyść. W poprzednim przykładzie zmienna `n` jest parametrem funkcji zewnętrznej. A ponieważ funkcja wewnętrzna znajduje się w środowisku funkcji zewnętrznej, to również ma dostęp do `n`. Funkcja wewnętrzna ma dostęp do zmiennych z otaczającego ją środowiska. Blok funkcji wewnętrznej odgrywa rolę podobną do bloku `{}` w zwykłej pętli i instrukcji warunkowej. Ważna różnica polega na tym, że zmienne zadeklarowane w funkcji wewnętrznej są niedostępne w funkcji zewnętrznej. I zazwyczaj jest to bardzo korzystne.

Przekazywanie argumentów

Zdefiniowana wcześniej funkcja `noisy`, zawierająca swój argument w innej funkcji, ma pewien poważny defekt.

```
function noisy(f) {  
  return function(arg) {  
    console.log("wywołanie z ", arg);  
    var val = f(arg);  
    console.log("wywołano z ", arg, "- otrzymano", val);  
    return val;  
  };  
}
```

Jeśli `f` przyjmie więcej niż jeden argument, to użyje tylko pierwszego. Wprawdzie można by było dodać więcej argumentów do funkcji wewnętrznej (`arg1, arg2` itd.) i przekazać je do `f`, ale nie wiadomo, ile dokładnie byłoby ich potrzeba. Poza tym takie rozwiązanie pozbawiłoby nas możliwości wykorzystania w `f` informacji z `arguments.length`. Jako że zawsze byłaby przekazywana taka sama liczba argumentów, nie byłoby wiadomo, ile w rzeczywistości zostało ich oryginalnie podanych.

Rozwiązaniem takich problemów w języku JavaScript jest użycie metody `apply`. Przekazuje się jej tablicę (lub obiekt podobny do tablicy) argumentów i wywołuje ona funkcję z tymi argumentami.

```
function transparentWrapping(f) {  
  return function() {  
    return f.apply(null, arguments);  
  };  
}
```

Jest to bezużyteczna funkcja, ale ilustruje interesującą nas technikę — zwracana przez nią funkcja przekazuje wszystkie podane argumenty, i tylko te, do `f`. Robi to, przekazując swój własny obiekt `arguments` do metody `apply`. Pierwszy argument metody `apply`, dla którego w tym przypadku przekazujemy `null`, może zostać wykorzystany do symulowania wywołania metody. Wróć do tego w następnym rozdziale.

JSON

Funkcje wyższego rzędu, które w jakiś sposób stosują funkcję do elementów tablicy, są powszechnie wykorzystywane w języku JavaScript. Jedną z najprostszych z nich jest metoda `forEach`. Istnieje też parę innych podobnych metod tablicowych. Aby się z nimi zaznajomić, poznamy nowy sposób przechowywania danych.

Kilka lat temu pewna osoba przeczesła mnóstwo archiwów i napisała książkę o historii mojego nazwiska (*Haverbeke* znaczy „owsiany potok”). Otwierając ją, liczyłem, że znajdę opowieści o rycerzach, piratach, alchemikach... ale okazało się, że pełno w niej historii flamandzkich wieśniaków. Dla zabawy zapisałem informacje o bezpośrednich przodkach w formacie komputerowym.

Zawartość utworzonego przeze mnie pliku wygląda mniej więcej tak:

```
[
  { "name": "Emma de Milliano", "sex": "f",
    "born": 1876, "died": 1956,
    "father": "Petrus de Milliano",
    "mother": "Sophia van Damme" },
  { "name": "Carolus Haverbeke", "sex": "m",
    "born": 1832, "died": 1905,
    "father": "Carel Haverbeke",
    "mother": "Maria van Brussel" },
  ... itd.
]
```

Format ten nazywa się JSON (wym. jak ang. imię Jason), czyli JavaScript Object Notation. Jest on powszechnie wykorzystywany do przechowywania danych i przesyłania ich przez internet.

JSON jest podobny do tego, jak w języku JavaScript zapisywane są tablice i obiekty, tylko ma pewne ograniczenia. Wszystkie nazwy własności muszą znajdować się w podwójnych cudzysłowach i można zapisywać tylko proste dane — żadnych funkcji, zmiennych ani niczego, co wymaga wykonania obliczeń. Także komentarze w formacie JSON są niedozwolone.

W języku JavaScript dostępne są funkcje `JSON.stringify` i `JSON.parse` służące do konwersji danych z tego formatu i na ten format. Pierwsza pobiera wartość JavaScript i zwraca łańcuch w formacie JSON. Druga pobiera taki łańcuch i zamienia go na wartość.

```
var string = JSON.stringify({name: "X", born: 1980});
console.log(string);
// → {"name":"X","born":1980}
console.log(JSON.parse(string).born);
// → 1980
```

Zmienna `ANCESTRY_FILE`, dostępna w środowisku testowym do tego rozdziału i w pliku pobranym pod adresem <ftp://ftp.helion.pl/przyklady/zrojsc.zip>, zawiera treść mojego pliku JSON w postaci łańcucha. Zdekodujemy ją i zobaczymy, ile osób zawiera.

```
var ancestry = JSON.parse(ANCESTRY_FILE);
console.log(ancestry.length);
// → 39
```

Filtrowanie tablicy

Aby znaleźć w zbiorze danych o przodkach osoby, które były młode w 1924 roku, można użyć poniższej funkcji. Odfiltrowuje ona elementy z tablicy, które nie przejdą testu.

```
function filter(array, test) {
  var passed = [];
  for (var i = 0; i < array.length; i++) {
    if (test(array[i]))
      passed.push(array[i]);
  }
  return passed;
}

console.log(filter(ancestry, function(person) {
  return person.born > 1900 && person.born < 1925;
}));
// → [{name: "Philibert Haverbeke", ...}, ...]
```

W funkcji tej został użyty argument o nazwie `test`. Jest to wartość funkcyjna mająca za zadanie zapełnić „lukę” w funkcjonalności. Funkcja `test` jest wywoływana dla każdego elementu, a jej wartość zwrrotna określa, czy element ma zostać dodany do tablicy zwrótniej.

Trzy osoby z pliku żyły i były młode w 1924 roku — moja babcia, mój dziadek i moja cioteczna babka.

Zwróć uwagę, że funkcja `filter` nie usuwa elementów z istniejącej tablicy, tylko tworzy nową tablicę zawierającą tylko te elementy, które przejdą test. Jest to funkcja *czysta*, ponieważ nie modyfikuje przekazanej jej tablicy.

Podobnie jak `forEach` funkcja `filter` też jest standardową metodą tablic. W powyższym przykładzie przedstawiłem jej definicję tylko po to, by pokazać, jak dokładnie działa. Ale od tej pory będę jej używać w następujący sposób:

```
console.log(ancestry.filter(function(person) {
  return person.father == "Carel Haverbeke";
}));
// → [{name: "Carolus Haverbeke", ...}]
```

Przekształcanie tablic za pomocą metody `map`

Powiedzmy, że mamy tablicę obiektów reprezentujących ludzi utworzoną poprzez przefiltrowanie w jakiś sposób tablicy `ancestry`. Wolelibyśmy jednak mieć tablicę nazwisk, ponieważ jest dla nas czytelniejsza.

Metoda `map` przekształca tablicę, stosując funkcję do wszystkich jej elementów i tworząc nową tablicę ze zwróconych wartości. Ta nowa tablica ma taką samą długość jak poprzednia, ale jej zawartość będzie „zmapowana” na nową formę przez funkcję.

```
function map(array, transform) {
  var mapped = [];
  for (var i = 0; i < array.length; i++)
    mapped.push(transform(array[i]));
  return mapped;
}

var overNinety = ancestry.filter(function(person) {
  return person.died - person.born > 90;
});
console.log(map(overNinety, function(person) {
  return person.name;
}));
// → ["Clara Aernoudts", "Emile Haverbeke", "Maria Haverbeke"]
```

Co ciekawe, osoby, które żyły nie mniej niż 90 lat, to te same, które widzieliśmy wcześniej — były młode w latach 1920. Jest to zarazem najmłodsza generacja w moim zbiorze danych. Podejrzewam, że ma to związek z postępem medycyny.

Podobnie jak `forEach` i `filter`, również `map` jest standardową metodą tablic.

Podsumowywanie przy użyciu metody reduce

Kolejną często wykonywaną czynnością na tablicach jest obliczanie z ich treści pojedynczej wartości. Jako przykład może posłużyć znany nam już przypadek sumowania kolekcji liczb. Innym przykładem może być znajdowanie najstarszej osoby w zbiorze danych.

Operacja wyższego rzędu reprezentująca tę technikę nazywa się **redukcją** (a czasami **zwijaniem**). Można ją sobie wyobrażać jak zwijanie tablicy po jednym elemencie. Przy sumowaniu liczb należy zacząć od zera i każdy kolejny element dodawać do aktualnej sumy.

Parametrami funkcji redukcyjnej `reduce` są, oprócz tablicy, funkcja łącząca i wartość początkowa. Funkcja ta jest nieco bardziej skomplikowana od `filter` i `map`, więc dobrze się jej przyjrzyj.

```
function reduce(array, combine, start) {
  var current = start;
  for (var i = 0; i < array.length; i++)
    current = combine(current, array[i]);
  return current;
}

console.log(reduce([1, 2, 3, 4], function(a, b) {
  return a + b;
}, 0));
// → 10
```

Standardowa tablicowa metoda `reduce`, która oczywiście jest podobna do powyższej, ma jeszcze jedno udogodnienie. Jeżeli tablica zawiera przynajmniej jeden element, można opuścić argument `start`. Wówczas metoda potraktuje pierwszy element tablicy jako wartość startową i rozpocznie redukowanie od drugiego elementu.

Aby za pomocą metody `reduce` znaleźć najstarszego z moich przodków, można napisać taki kod:

```
console.log(ancestry.reduce(function(min, cur) {
  if (cur.born < min.born) return cur;
  else return min;
}));
// → {name: "Pauwels van Haverbeke", born: 1535, ...}
```

Składalność

Zastanów się, jak można by było napisać poprzedni przykład (znajdujący najstarszą osobę) bez użycia funkcji wyższego rzędu. Odpowiedni kod nie byłby aż taki zły.

```
var min = ancestry[0];
for (var i = 1; i < ancestry.length; i++) {
  var cur = ancestry[i];
  if (cur.born < min.born)
    min = cur;
}
console.log(min);
// → {name: "Pauwels van Haverbeke", born: 1535, ...}
```

Znajduje się w nim parę zmiennych więcej i program ten jest o dwa wiersze dłuższy od poprzedniego, ale też jest zrozumiały.

Zalety funkcji wyższego rzędu stają się ewidentne, gdy trzeba **składać** funkcje. W ramach przykładu napiszemy kod znajdujący średni wiek mężczyzn i kobiet w zbiorze danych.

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}
function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }

console.log(average(ancestry.filter(male).map(age)));
// → 61.67
console.log(average(ancestry.filter(female).map(age)));
// → 54.56
```

(To trochę dziwne, że trzeba zdefiniować funkcję `plus`, ale w języku JavaScript operatory, w odróżnieniu od funkcji, nie są wartościami, więc nie mogą być przekazywane jako argumenty).

Zamiast wielkiej pętli jest tu logika programu starannie złożona z interesujących nas koncepcji — określanie płci, obliczanie wieku i uśrednianie wartości. Możemy je zastosować jedna po drugiej, aby otrzymać szukany wynik.

Jest to *doskonale* sposób pisania przejrzystego kodu. Niestety nie ma nic za darmo.

Koszty

W szczęśliwej krainie eleganckiego kodu i pięknych tęczy mieszka złośliwy potwór o nazwie **niewydajność**.

Program przetwarzający tablicę w najelegantszej formie składa się z sekwencji osobnych kroków, z których każdy robi coś z tą tablicą i tworzy nową tablicę. Ale tworzenie takiej ilości pośrednich tablic jest kosztowne.

Analogicznie przekazanie funkcji do metody `forEach`, aby wykonała iterację za nas, jest wygodnym i czytelnym rozwiązaniem. Ale wywołania funkcji w JavaScriptcie są bardzo kosztowne w porównaniu z wykonywaniem zwykłych pętli.

To samo dotyczy wielu innych technik pozwalających zwiększyć czytelność programu. Abstrakcje dokładają warstwę między to, co robi komputer, a koncepcje, z którymi pracuje programista, i w ten sposób dodają maszynie pracy. Nie jest to oczywiście żelazna zasada — istnieją języki programowania z lepszym wsparciem dla abstrakcji, w których nie wiążą się one z pogorszeniem wydajności, a nawet w JavaScriptcie doświadczony programista potrafi znaleźć sposoby na pisanie bardziej abstrakcyjnego, choć nadal szybkiego kodu. Niemniej jednak problem ten istnieje.

Na szczęście większość komputerów jest niesamowicie szybka. Jeśli więc masz do przetworzenia w miarę nieduży zbiór danych albo robisz coś, co musi mieścić się w ludzkiej skali czasu (np. po każdym kliknięciu przycisku), to *nie ma znaczenia*, czy napiszesz piękne rozwiązanie działające w pół milisekundy, czy doskonale zoptymalizowane zajmujące jedną dziesiątą milisekundy.

Dobrym pomysłem jest oszacowanie, jak często ma być wykonywany dany program. Jeśli wstawisz pętlę do pętli (bezpośrednio albo przez wywołanie funkcji wykonującej pętlę), to kod wewnętrznej pętli zostanie wykonany $N \times M$ razy, gdzie N jest liczbą powtórzeń pętli zewnętrznej, a M liczbą powtórzeń pętli wewnętrznej w każdej iteracji pętli zewnętrznej. Jeśli w tej wewnętrznej pętli umieścisz jeszcze kolejną wykonującą P iteracji, to jej instrukcje zostaną wykonane $N \times M \times P$ razy itd. W sumie może wyjść bardzo dużo obliczeń i jeśli program będzie powolny, problem może sprowadzać się do jednego miejsca w pętli wewnętrznej.

Prapraprapra...

W pliku z danymi znajdują się informacje o moim dziadku Philibercie Haverbeke. Wychodząc od niego, mogę prześledzić swój rodowód, aby dowiedzieć się, czy pochodzę od swojego najstarszego przodka Pauwelsa van Haverbeke w linii prostej. Jeśli tak, to chciałbym wiedzieć, ile teoretycznie mam jego kodu DNA.

Aby móc przechodzić od imienia i nazwiska rodzica do obiektu reprezentującego tę osobę, najpierw zbudujemy obiekt wiążący imiona z ludźmi.

```
var byName = {};  
ancestry.forEach(function(person) {  
    byName[person.name] = person;  
});  
  
console.log(byName["Philibert Haverbeke"]);  
// → {name: "Philibert Haverbeke", ...}
```

Jednak problem nie jest tak prosty, że wystarczy przejść po własnościach `father` i policzyć, ile stopni się minęło przed dotarciem do Pauwelsa. W rodzinie było parę przypadków małżeństw między dalszymi krewnymi (małe woski i te sprawy). To powoduje, że niektóre gałęzie łączą się z powrotem w paru miejscach, co z kolei oznacza, że mam więcej niż $1/2^G$ wspólnych genów z daną osobą (G oznacza liczbę pokoleń między Pauwelsem a mną). Wzór ten oznacza, że każde pokolenie dzieli pulę genów na pół.

Rozsądnym podejściem do tego problemu jest potraktowanie go w sposób analogiczny do redukcji, czyli operacji polegającej na sprowadzeniu tablicy do pojedynczej wartości poprzez obliczenie kombinacji jej wartości od lewej do prawej. W tym przypadku też chcemy skondensować strukturę danych do pojedynczej wartości, ale poprzez podążanie śladem więzów rodzinnych. *Kształt* tych danych przypomina drzewo, a nie płaską listę.

Redukcję tej struktury przeprowadzimy w ten sposób, że wartość dla danej osoby obliczymy przez połączenie wartości jej przodków. Można to zrobić rekurencyjnie — jeżeli interesuje nas osoba A , musimy obliczyć wartości dla jej rodziców, co z kolei wymaga obliczenia wartości dla dziadków osoby A itd. Teoretycznie mogłoby to doprowadzić do konieczności wykonania obliczeń dla nieskończonej liczby osób, ale ponieważ nasz zbiór danych jest skończony, musimy znaleźć jakiś sposób na zatrzymanie się. Dlatego dodamy możliwość przekazania do naszej funkcji redukcyjnej wartości domyślnej reprezentującej osoby, których nie ma w danych. W naszym przypadku wartością tą będzie po prostu zero, ponieważ wychodzimy z założenia, że osoby, których nie ma na liście, nie mają wspólnego DNA z interesującym nas przodkiem.

Funkcja `reduceAncestors` pobiera osobę, funkcję łączącą wartości z dwóch rodziców i wartość domyślną i kondensuje wartość z drzewa rodzinnego.

```
function reduceAncestors(person, f, defaultValue) {  
    function valueFor(person) {  
        if (person == null)  
            return defaultValue;  
        else  
            return f(person, valueFor(byName[person.mother]),  
                    valueFor(byName[person.father]));  
    }  
    return valueFor(person);  
}
```

Funkcja wewnętrzna `valueFor` obsługuje pojedynczą osobę. Dzięki magii rekurencji może wywołać samą siebie, aby obsłużyć ojca i matkę tej osoby. Wyniki, wraz z samym obiektem osoby, są przekazywane do funkcji `f`, która zwraca rzeczywistą wartość dla tej osoby.

Później przy użyciu tej wartości można obliczyć, ile wspólnego DNA miała moja babcia z Pauwelsem van Haverbeke, i podzielić to przez cztery.

```
function sharedDNA(person, fromMother, fromFather) {
  if (person.name == "Pauwels van Haverbeke")
    return 1;
  else
    return (fromMother + fromFather) / 2;
}
var ph = byName["Philibert Haverbeke"];
console.log(reduceAncestors(ph, sharedDNA, 0) / 4);
// → 0.00049
```

Osoba nazywająca się Pauwels van Haverbeke oczywiście ma 100 procent wspólnych genów z Pauwelsem van Haverbeke (w zbiorze danych nie ma osób, które nazywały się dokładnie tak samo), więc dla niego funkcja zwraca 1. Wszyscy pozostali ludzie mają z nim wspólną średnią wartość z wartości dzielonych przez ich rodziców.

Statystycznie rzecz ujmując, mam około 0,05% wspólnego DNA z przodkiem, który żył w XVI wieku. Oczywiście to tylko statystyczne przybliżenie, a nie dokładna liczba. Nie jest to wiele, ale biorąc pod uwagę ilość materiału genetycznego w człowieku (około 3 miliardów par zasad), możliwe, że w moim organizmie wciąż istnieje jakaś cząstka pochodząca od Pauwelsa.

Wartość tę można by było też obliczyć bez posługiwania się funkcją `reduceAncestors`. Ale oddzielenie ogólnego podejścia (kondensacja drzewa rodzinnego) od specyficznego przypadku (obliczenie ilości wspólnego DNA) może poprawić klarowność kodu i pozwala na wielokrotne wykorzystanie abstrakcyjnej części programu w innych przypadkach. Na przykład poniższy kod znajduje odsetek znanych przodków wybranej osoby, którzy żyli dłużej niż 70 lat.

```
function countAncestors(person, test) {
  function combine(person, fromMother, fromFather) {
    var thisOneCounts = test(person);
    return fromMother + fromFather + (thisOneCounts ? 1 : 0);
  }
  return reduceAncestors(person, combine, 0);
}
function longLivingPercentage(person) {
  var all = countAncestors(person, function(person) {
    return true;
  });
  var longLiving = countAncestors(person, function(person) {
```

```
    return (person.died - person.born) >= 70;
  });
  return longLiving / all;
}
console.log(longLivingPercentage(byName["Emile Haverbeke"]));
// → 0.145
```

Zważywszy na to, że nasz zbiór danych zawiera dość przypadkową kolekcję osób, liczb takich nie należy traktować zbyt poważnie. Ale przedstawiony kod ilustruje fakt, że funkcja `reduceAncestors` stanowi dla nas przydatne słowo w pracy z drzewami rodzinnymi.

Wiązanie

Metoda `bind`, którą mają wszystkie funkcje, tworzy nową funkcję wywołującą oryginalną funkcję, ale z ustawionymi niektórymi argumentami.

Poniżej znajduje się przykład użycia metody `bind`. Jest to definicja funkcji o nazwie `isInSet`, sprawdzającej, czy dana osoba znajduje się w podanym zbiorze łańcuchów. Aby wywołać metodę `filter` w celu zebrania obiektów tych osób, których imienia i nazwiska znajdują się w określonym zbiorze, możemy napisać wyrażenie funkcyjne wywołujące funkcję `isInSet` z naszym zbiorem jako pierwszym argumentem albo *częściowo zastosować* funkcję `isInSet`.

```
var theSet = ["Carel Haverbeke", "Maria van Brussel", "Donald Duck"];
function isInSet(set, person) {
  return set.indexOf(person.name) > -1;
}
console.log(ancestry.filter(function(person) {
  return isInSet(theSet, person);
}));
// → [{name: "Maria van Brussel", ...},
//     {name: "Carel Haverbeke", ...}]
console.log(ancestry.filter(isInSet.bind(null, theSet)));
// → ... taki sam wynik
```

Wywołanie funkcji `bind` zwróci funkcję, która wywoła funkcję `isInSet` ze zbiorem `theSet` jako pierwszym argumentem i pozostałymi argumentami przekazanymi do powiązanej funkcji.

Pierwszy argument, w którego miejscu w przykładzie przekazano wartość `null`, jest używany dla wywołań metody, podobnie jak pierwszy argument funkcji `apply`. Bardziej szczegółowo opisuję to w następnym rozdziale.

Podsumowanie

Możliwość przekazywania funkcji jako wartości do innych funkcji to nie tylko sztuczka, ale bardzo przydatna właściwość języka JavaScript. Pozwala na pisanie obliczeń jako funkcji z „lukami”, które kod wywołujący musi wypełnić, dostarczając wartości funkcyjne implementujące braki.

Tablice udostępniają parę metod wyższego rzędu — `forEach` do wykonywania działań na każdym elemencie tablicy po kolei, `filter` do tworzenia nowych tablic z odfiltrowaną częścią elementów, `map` do tworzenia nowych tablic, których każdy element został przepuszczony przez jakąś funkcję, oraz `reduce` do łączenia wszystkich elementów tablicy w jedną wartość.

Funkcje mają metodę `apply`, przy użyciu której można je wywoływać z tablicą argumentów. Mają też metodę `bind` służącą do tworzenia częściowo zastosowanych wersji funkcji.

Ćwiczenia

Splaszczanie

Za pomocą metod `reduce` i `concat` „splaszcz” tablicę tablic do postaci pojedynczej tablicy zawierającej wszystkie elementy tablic wejściowych.

Różnica wieku między matką i dzieckiem

Przy użyciu przykładowego zbioru danych z tego rozdziału oblicz średnią różnicę wieku między matkami i dziećmi (wiek matki w chwili urodzenia dziecka). Możesz użyć zdefiniowanej w tym rozdziale funkcji `average`.

Zwróć uwagę, że nie wszystkie matki znajdujące się w danych są obecne w tablicy. W sytuacji tej może być przydatny obiekt `byName`, który ułatwia znajdowanie obiektu osoby po jej imieniu i nazwisku.

Historyczna średnia długość życia

Gdy poszukaliśmy w naszym zbiorze danych osób, które żyły dłużej niż 90 lat, znaleźliśmy tylko osoby z ostatniego pokolenia. Przyjrzyjmy się bliżej temu zjawisku.

Oblicz i wyświetl średni wiek osób ze zbioru danych dla każdego stulecia. Wiek, w którym żyła dana osoba, można obliczyć, dzieląc datę jej śmierci przez 100 i zaokrąglając otrzymany wynik w górę, np. `Math.ceil(person.died / 100)`.

Zadanie dla chętnych: napisz funkcję o nazwie `groupBy` abstrahującą operację grupowania. Funkcja ta powinna przyjmować jako argumenty tablicę i funkcję obliczającą grupę dla elementu w tej tablicy oraz zwracającą obiekt odwzorowujący nazwy grup na tablice numerów grup.

Wszystko i trochę

Tablice mają też standardowe metody o nazwach `every` i `some`. Obie one przyjmują funkcję predykatywną, która dla argumentu w postaci elementu tablicy zwraca prawdę albo fałsz. Podobnie jak operator `&&` zwraca prawdę tylko wtedy, gdy wyrażenia po jego obu stronach są prawdziwe, tak funkcja `every` zwraca prawdę tylko wtedy, gdy predykat zwraca prawdę dla *wszystkich* elementów tablicy. Natomiast funkcja `some` zwraca prawdę, gdy predykat zwróci prawdę dla *któregokolwiek* z elementów. Funkcje te nie przetwarzają więcej argumentów, niż jest konieczne — na przykład jeśli funkcja `some` znajdzie predykat zwracający prawdę dla pierwszego elementu tablicy, to nie sprawdza już pozostałych wartości.

Napisz dwie funkcje, `every` i `some`, działające jak te metody, tylko niebędące metodami, a pobierające tablicę jako swój pierwszy argument.

Problem z językami obiektowymi polega na tym, że wszystkie działają w niejawnie powiązanych z nimi środowiskach, których nie da się pominąć. Chcesz banana, ale w efekcie otrzymujesz goryla trzymającego banana i całą dżunglę.

— Joe Armstrong, w wywiadzie dla *Sztuki kodowania*

Skorowidz

A

- abstrahowanie żądań, 343
- abstrakcje, 106, 246
- adres
 - IP, 237
 - URL, 237
 - URL danych, 376
- aktualizowanie serwera, 424
- aktywacja, 276, 355
- AMD, asynchronous module definition, 211
- analiza, 93
- analiza składni, 219
- animowanie, 153, 259
- aplikacje, 220
- aplikacje profilujące, 436
- argument, 47
- argumenty opcjonalne, 70
- arkusze stylów, 257, 258
- arytmetyka, 34
- asercje, 176
- asynchroniczne wejście, 388
- asynchroniczność, 389
- asynchroniczny interfejs, 388
- atrybut, 239, 251
 - href, 376
 - style, 256
 - xmlns, 310
- automatyczna konwersja typów, 39

B

- bezpieczeństwo, 348
- biblioteki, 205
- bit, 31
- blok, 52, 56, 134
 - catch, 174
 - finally, 173
 - try, 173
- błąd 404, 399, 414
- błędy, 165, 402
- błędy programisty, 165
- bug, 165

C

- CSS, Cascading Style Sheets, 257
- czas wykonywania skryptu, 277

Ć

- ćwiczenia
 - automatyczne uzupełnianie, 367, 462
 - budowa tabeli, 262, 457
 - cenzura klawiatury, 282, 458
 - drapieźniki, 163, 453
 - elementy według nazwy znacznika, 262, 457
 - FizzBuzz, 59, 448
 - gra w życie Conwaya, 367, 463
 - historyczna średnia długość życia, 120, 451
 - interfejs sekwencyjny, 142, 452
 - jeszcze raz liczby, 202, 454

ćwiczenia
kapelusze kota, 263
karty, 282, 458
kolejna komórka, 142, 452
komentarze, 231, 456
koniec gry, 306, 459
kształty, 331, 460
lepsze szablony, 427, 467
liczenie znaków, 79, 449
lista, 102, 450
mierzenie czasu, 445
minimum, 78, 448
naprawienie zakresu, 231, 457
nazwy miesięcy, 216, 455
negocjacja treści, 350, 461
negocjacja treści raz jeszcze, 404, 465
obliczenia na zapas, 332, 461
oczekiwanie na wiele obietnic, 350, 462
odbijająca się piłka, 332, 461
odwracanie tablicy, 101, 449
optymalizacja, 446, 468
pętlowy trójkąt, 59, 447
plansza do gry w szachy, 59, 448
porównywanie głębokie, 102, 450
powrót do elektronicznego życia, 216, 455
pracownia JavaScript, 367, 462
prostokąty, 380, 463
próbki kolorów, 381, 464
publiczna przestrzeń w internecie, 406, 466
rekurencja, 78, 448
resetowanie pól komentarzy, 427, 467
rodzaje cudzysłowów, 202, 454
różnica wieku między matką i dzieckiem, 120, 451
spłaszczanie, 120
spróbuj jeszcze raz, 177, 453
suma przedziału liczb, 101, 449
sztuczna głupota, 162, 453
tablice, 230, 456
tamowanie wycieku, 405, 465
trop myszy, 282, 458
tworzenie katalogów, 405, 466
typ wektorowy, 141, 452
wstrzymywanie gry, 306, 459
wszystko i trochę, 121, 451
wykluczeni ze skryptów, 428, 467
wykres kołowy, 331, 460
wypełnianie zalewowe, 382, 464
wyrażeniowy golf, 201
zależności cykliczne, 217, 456
zamknięcie, 230, 456

zamknięte pudełko, 177, 454
zapisywanie danych na dysku, 427, 466
znajdowanie drogi, 444, 468

D

dane XML, 341
debouncing, 280
debugowanie, 166, 168
definiowanie
funkcji, 64
grafu, 432
deklaracje, 256
deklarowanie funkcji, 67, 207
długie sondowanie, 409, 416
dokument HTML, 238, 243
DOM, document object model, 244
domena, 237
dopasowania, 186, 189
dopasowywanie
wzorców, 182
zbiorów znaków, 183
drzewa, 245
drzewo składniowe, 221
dynamiczne tworzenie obiektów, 195
działania domyślne, 269
działanie programu, 23
dziedziczenie, 139

E

efekt uboczny, 44
egzemplarz, 127
elementy
blokowe, 253
śródliniowe, 253
encja, 239
etykieta case, 56
ewaluator, 223

F

filtrowanie tablicy, 112
fokus, 355
format
JSON, 111, 342, 346
XML, 339
formatowanie elementów, 257
formowanie tabeli, 132
formularze, 337, 353

- funkcja, 47, 63, 228
 - addEventListener, 281
 - alert, 48
 - animate, 261
 - apply, 119
 - assert, 176
 - branch, 323
 - cancelAnimationFrame, 279
 - colWidths, 134
 - confirm, 48
 - console.log, 47
 - dataTable, 139
 - dayName, 206
 - define, 212, 216
 - defineProperty, 138
 - drawFrame, 325
 - drawLine, 135
 - drawRow, 134
 - evaluate, 224
 - filter, 113
 - findDate, 188
 - flipHorizontally, 322
 - forEach, 113
 - getChangedTalks, 418
 - getJSON, 347
 - getModule, 212, 213
 - getURL, 345
 - hasEvent, 92
 - highlightCode, 252
 - isInSet, 119
 - isNaN, 50
 - JSON.parse, 111
 - JSON.stringify, 111
 - lastElement, 176
 - Math.cos, 260
 - Math.max, 98, 326
 - Math.random, 100
 - Math.sqrt, 91
 - Number, 49
 - Object.defineProperty, 138
 - Object.getPrototypeOf, 127
 - parseApply, 223
 - parseExpression, 223
 - parseExpressions, 222
 - postMessage, 278
 - prompt, 49
 - promptDirection, 175
 - randomElement, 149
 - randomPointInRadius, 380
 - readFile, 393, 394
 - reduce, 134
 - reduceAncestors, 117, 118
 - registerChange, 415, 418
 - repeat, 135
 - requestAnimationFrame, 260, 277, 341
 - require, 209–211, 216, 391
 - rowHeights, 134
 - setInterval, 379
 - setTimeout, 279, 280
 - skipSpace, 222
 - stopPropagation, 268
 - test, 112
 - treeGraph, 432
 - valueFor, 118
 - waitForChanges, 421
 - whenDepsLoaded, 213
 - withContext, 172
- funkcje
 - argumenty opcjonalne, 70
 - czyste, 77
 - definicja, 64
 - deklaracja, 68
 - dołączające, 209
 - jako procedury do obsługi, 266
 - jako wartości, 67
 - obsługi czynności, 158
 - parametry, 65
 - przekazywanie argumentów, 110
 - rekurencja, 72
 - składanie, 115
 - skutki uboczne, 77
 - stos wywołań, 68
 - wiązanie, 119
 - wyższego rzędu, 105, 109
 - zakresy dostępności, 65
 - zamknięcia, 71
 - zwrotne, 404

G

- gniazda sieciowe, web sockets, 409
- gorący kod, 430
- gra platformowa, 285
 - aktorzy, 289, 299
 - czynności, 299
 - hermetyzacja, 291
 - kolizje, 297
 - poziomy, 287
 - ruch, 297
 - rysowanie, 292
 - śledzenie klawiszy, 303

gra platformowa
 technologia, 286
 uruchamianie, 303
 wczytywanie poziomu, 288
graf, 431
graf drzewiasty, 433
grafika
 bitmapowa, 319
 wektorowa, 309
granice słów i łańcuchów, 188
grawitacja, 301
grupowanie podwyrażeń, 185
grupy, 186

H

hermetyzacja, 124, 291
hierarchia operatorów, 34
HTML
HTML, Hypertext Markup Language, 238,
 240, 419
HTTP, Hypertext Transfer Protocol, 237,
 335–348, 410
HTTPS, 348

I

implementacja AMD, 214
indeks, 83
instalowanie modułów, 392
instrukcja, 44
instrukcja switch, 56
interfejs, 207
 graficzny, 329
 HTTP, 410
 modułu, 205, 214
 niskopoziomowy, 215
 programistyczny, 148
 warstwowy, 215
 wysokopoziomowy, 215
 XMLHttpRequest, 339, 349
interferencja prototypów, 129
internet, 236

J

JavaScript, 24
jednostka
 %, 276
 em, 261
 px, 261

język
 HTML, 238
 JavaScript, 24
 programowania, 219
 specjalistyczny, 230
JSON, 111

K

kanwa, 309, 311
kaskadowe arkusze stylów, 257
kasowanie przekształceń, 323
klatka, 320
klauzula default, 56
klawisze modyfikujące, 271
klient, 236, 364, 419
kliknięcia, 272
kod, 25
kodowanie
 URL, 338
 UTF-8, 397
kolizje, 297
komentarze, 57
kompilacja, 229, 429
kompilacja etapowa, 430
konstrukcja
 do, 225
 if, 225
 new Function, 227
 while, 23, 229
konstruktor, 127
 Array, 147, 227
 BouncingCritic, 149
 FileReader, 363
 Function, 209
 LifelikeWorld, 157
 Person, 167
 RegExp, 195
 World, 148, 151
 XMLHttpRequest, 339
kontekst, 311
konwersja typów, 39
korelacja, 89, 90
koszty, 116
krzywe, 314

L

liczby, 32
 binarne, 31
 całkowite, 33

- specjalne, 34
- ułamkowe, 33

lista przemów, 421

liść, 245

L

ładowanie strony, 277

łańcuch zapytania, 338

łańcuchy, 35, 96

luk, 315

M

metoda, 84, 125

- act, 150, 154, 156
- addEventListener, 266, 267
- appendChild, 249
- apply, 125
- bind, 119
- call, 125
- charAt, 97
- cloneNode, 424
- concat, 96
- console.log, 390
- document.createElement, 250
- drawFrame, 324
- drawImage, 319, 324, 327
- drawPlayer, 328
- encodeURIComponent, 338
- figlet.text, 393
- fillRect, 312, 320
- find, 148
- findAll, 149
- forEach, 116, 151
- getAttribute, 251
- getBoundingClientRect, 254
- getContext, 311, 330
- getDay, 206
- getElementByTagName, 248
- indexOf, 92
- insertBefore, 249
- join, 85
- lastIndexOf, 95
- letAct, 153, 157
- map, 113, 152
- open, 340
- pop, 85, 95
- preventDefault, 269, 372
- push, 85, 95

- querySelector, 259
- querySelectorAll, 258, 259
- reduce, 114
- removeEventListener, 267
- replace, 192, 193
- replaceChild, 249
- scale, 321
- scrollPlayerIntoView, 295
- search, 195
- setAttribute, 251
- setRequestHeader, 340
- slice, 96, 250
- test, 201
- times, 437
- toString, 126, 132
- translate, 321
- trim, 97
- turn, 153
- updateViewport, 326

metody, 84

- pobierające, 137
- ustawiające, 137
- żądania, 336

mikrooptymalizacja, 438

model DOM, 244, 246, 261

moduł, 203, 205, 216, 391

- fs, 394
- HTTP, 395
- systemu plików, 393
- weekDay, 212

moduły

- CommonJS, 211
- z NPM, 392

modyfikowanie

- dokumentu, 249
- zmiennych, 55

N

nagłówek, 337

narzędzia

- do mierzenia czasu, 436
- do rysowania, 372

nasłuchiwanie, 236

nawias ostry, 239

nazwy zmiennych, 57

Node.js, 387

notacja, 219

notacja naukowa, 33

NPM, 392, 399



- obiekt, 81, 85, 123
 - arguments, 97
 - Date, 206
 - exports, 210
 - FileReader, 363
 - globalny, 100
 - Grid, 147
 - Math, 98
 - obietnicy, 345
 - RegExp, 195
 - SecurityError, 377
 - sessionStorage, 366
 - specialForms, 224
 - World, 149
- obiektyowy model dokumentu, 243, 244
- obiekty
 - bez prototypów, 131
 - jako interfejsy, 207
 - jako słowniki, 92
 - prototypowe, 126
 - zdarzeń, 267
- obietnice, 345
- obliczanie
 - korelacji, 90
 - siły, 433
- obramowanie akapitu, 254
- obrazy, 319
- obrót, 321
- obrys, 312
- obsługa
 - błędów, 165, 402
 - kolizji, 302
 - niepowodzeń, 343
 - wyjątków, 171
 - wyrażeń, 224
 - zdarzeń, 265
- odbicie, 322
- odnośnik, 376
- odśmiecanie pamięci, 441
- odwijanie stosu, 171
- odwrotny serwer proxy, 412
- operacja
 - console.log, 23
 - range, 24
 - sum, 24
- operator, 34
 - &&, 38, 41
 - ||, 38
 - ==, 88

- delete, 87
- eval, 209
- in, 87
- instanceof, 140, 175
- modulo, 34
- nie, 38
- trójargumentowy, 38
- typeof, 36
- warunkowy, 39
- operatory
 - dwuargumentowe, 36
 - jednoargumentowe, 36
 - logiczne, 38, 40
 - powtórzeniowe, 194
- opóźnienie, 280
- optymalizacja, 430
- optymalizacja tworzenia obiektów, 440
- ożywianie świata, 161

P

- parametry funkcji, 64, 65
- parser, 219
- pędzel, 374
- pętla
 - do, 51
 - for, 54, 438
 - inicjowanie, 54
 - kończenie działania, 55
 - nieskończona, 55
 - while, 51
 - wyrażenie sprawdzające, 54
- piaskownica, 241
- piaskownica dla HTTP, 342
- plik garble.js, 391
- pliki INI, 198
- pobieranie danych XML, 341
- pokazywanie zmian, 426
- pokoleniowe odśmiecanie pamięci, 441
- pola
 - formularza, 354, 366
 - opcji do wyboru, 360
 - plikowe, 362
 - tekstowe, 358
 - wyboru, 359
- polecenie node, 389
- polimorfizm, 132
- populacja nowego świata, 160
- porównywanie, 37
- port, 236

- powtarzanie części wzorca, 184
- pozycjonowanie, 259
- procedury obsługi zdarzeń, 265
- profilowanie, 435
- program, 21, 43
 - Browserify, 211
 - rysunkowy, 369
 - implementacja, 370
 - model DOM, 370
 - pędzle, 374
 - podstawa, 371
 - wczytywanie obrazów, 377
 - wybór narzędzi, 372
 - wykończenie, 379
 - zapisywanie, 376
- programowanie, 20
 - asynchroniczne, 305, 349
 - literackie, 204
 - obiektywne, 140
- projekt
 - elektroniczne życie, 145
 - gra platformowa, 285
 - język programowania, 219
 - program rysunkowy, 369
 - serwis dla pasjonatów, 407
- projektowanie interfejsu, 214
- propagacja
 - błędów, 170
 - zdarzeń, 268
- protokół
 - HTTP, 237, 335
 - HTTPS, 349
 - sieciowy, 236
 - TCP, 236
- prototypy, 126
- przechwytywanie wyjątków, 174
- przeglądanie
 - dopasowanych elementów, 197
 - drzewa, 246, 248
- przeglądarka, 233, 235
 - Chrome, 242
 - Firefox, 242
 - Internet Explorer, 241
 - Mosaic, 241
 - Safari, 242
- przekazywanie argumentów, 110
- przekształcanie tablic, 113
- przekształcenia, 320
- przemowy, 414
- zapełnienie, 33

- przesłanie dziedziczonych własności, 128
- przestrzenie nazw, 98, 204, 206
- przesunięcie, 321
- przetwarzanie plików INI, 198
- przewidywalność, 214
- przewijanie, 275
- przyciski radiowe, 359
- pseudoselektor :hover, 274
- punkt wstrzymania, 170

R

- redukcja, 114
- reguły CSS, 257
- rekurencja, 72
- relacje między modułami, 205
- repozytorium NPM, 392
- reprezentacja przestrzeni, 146
- reszta z dzielenia, 34
- robienie notatek, 364
- robotnik sieciowy, 278
- rodzaje pól formularzy, 354, 366
- router, 412
- rozkład
 - grafu, 431
 - ukierunkowany siłowo, 433
- rozmiar pędzla, 375
- rozmieszczenie elementów, 253
- rozwijanie funkcji, 437
- RPC, remote procedure call, 348
- ruch, 155
- ruch myszy, 273
- rysowanie, 292
 - na kanwie, 309
 - wykresu kołowego, 317

S

- selektory, 258
- serwer, 236, 412
 - plików, 398
 - proxy, 412
- serwis dla pasjonatów, 407
- serwowanie plików, 413
- siatka, 146
- siatka komórek, 137
- sieci, 236
- sieć ogólnosiwiatowa, 237
- silniki
 - fizyki, 297
 - JavaScript, 429

- siła, 433
- skalowanie, 321
- składanie funkcji, 115
- składnia, 219
- składnia selektorów, 258
- skutki uboczne, 48, 77
- słowa zarezerwowane, 46
- słownik, 92
- słowo kluczowe
 - class, 253
 - function, 64
 - if, 50
 - return, 64
 - this, 125
 - var, 44
- sondowanie, 266
- specjalne konstrukcje, 224
- stos wywołań, 68
- stosowanie
 - wcięć, 53
 - wielkich liter, 57
- struktura
 - dokumentu, 243
 - programu, 43
- struktury danych, 81
- strumienie, 396
- styl CommonJS, 211
- style, 255
- SVG, Scalable Vector Graphics, 309, 330
- symbole zastępcze znaków, 35
- symulacja form życia, 157
- synchroniczne wejście, 388
- system
 - plików, 393
 - szablonów, 421
 - testowy, testing framework, 168

Ś

- ścieżka, 312
- ślad stosu, stack trace, 172
- śledzenie
 - czasu, 325
 - klawiszy, 303
- średnik, 54
- środowisko, 46, 226
- środowisko Node.js, 387

T

- tabele
 - formowanie, 132
- tablice, 83, 112
 - filtrowanie, 112
 - podsumowywanie, 114
 - przekształcanie, 113
- TCP, Transmission Control Protocol, 236
- tekst, 318
- testowanie, 167
- trasowanie, 412
- treść właściwa funkcji, 64, 337
- tryb ścisły, 166
- tworzenie
 - abstrakcji, 246
 - języka programowania, 219
 - modelu DOM, 370
 - obiektów RegExp, 195
 - piaskownicy, 241
 - ścieżki, 314
 - węzłów, 249
 - wyrażeń regularnych, 182
 - zadań idempotentnych, 401
- typ
 - CanvasDisplay, 324
 - Date, 187
 - integer, 33
 - logiczny, 37
 - łańcuch, 35
 - MIME, 399
 - Vector, 146
 - View, 154
- typy dynamiczne, 443

U

- uchwyt, 346
- układ grafowy, 431
- ukośnik, 57
- unikanie pracy, 438
- URL, Universal Resource Locator, 237
- uruchamianie gry, 303
- usługa NPM, 205
- ustawienie absolute, 259
- usuwanie nieużytków, 440

W

- wartości, 32
 - logiczne, 37
 - niezdefiniowane, 39
 - typu liczbowego, 32
 - zwrotne, 48
- wartość NaN, 40
- wątki
 - sterowania, 388
 - wykonawcze, 278
- wcięcia w kodzie, 53
- wczytywanie
 - modułów, 211
 - obrazów, 377
 - w tle, 211
 - z plików, 377
- wejście
 - asynchroniczne, 388
 - synchroniczne, 388
- węzeł, node, 245, 249, 404
- węzły
 - DOM, 266
 - modelu DOM, 246
 - potomne, 245
- wiązanie, 119
- wielokrotne wykorzystywanie kodu, 204
- wirtualny ekosystem, 145
- wirus, 241
- własności, 83
 - niewyliczalne, 130
 - wyliczalne, 130
- własność
 - background, 293
 - clientWidth, 254
 - documentElement, 244
 - files, 362
 - fillStyle, 312, 375
 - flipPlayer, 324
 - globalCompositeOperation, 374
 - lastIndex, 196
 - nextSibling, 247
 - offsetWidth, 254
 - parentNode, 246
 - position, 259
 - previousSibling, 247
 - responseText, 341
 - responseXML, 341
 - strokeStyle, 312, 375
 - transform, 309
 - wobble, 300

- własny język programowania, 219, 230
- wojny przeglądark, 241
- współczynnik phi, 89
- wybór interfejsu graficznego, 329
- wycofywanie, 190
- wydajność JavaScriptu, 429
- wyjątki, 171, 172
- wykonywanie
 - danych, 209
 - warunkowe, 50
- wykreś kołowy, 317
- wylączenie pól, 356
- wypełnienie, 312
- wyrażenia regularne, 181, 200
- wyrażenie, 43, 220
- wyrażenie funkcyjne, 207
- wysyłanie żądania, 339
- wyświetlacz, 292
- wyświetlanie przemów, 422
- wywoływanie funkcji, 47
- wyzwalanie zdarzeń, 280
- wzorce, 182, 184, 200
- wzorze wyboru, 189

X

- XMLHttpRequest, 339

Z

- zachłanność, 193
- zagnieżdżone zakresy dostępności, 66
- zakres dostępności, 65, 151
 - globalny, 208
 - leksykalny, 66
- zamknięcia, 71
- zapisywanie danych, 364, 441
 - przekształceń, 323
 - w obiektach, 441
- zasada
 - dopasowywania, 189
 - kolejności, 258
- zasób, 237
- zbiory danych, 82
- zdalne wywołania procedur, 348
- zdarzenia, 266
 - aktywacji, 276
 - klawiszy, 270
 - przewijania, 275

- zdarzenie
 - change, 359
 - dblclick, 272
 - error, 345
 - keydown, 270
 - keyup, 271
 - load, 277
 - mousedown, 268
 - mousemove, 273, 280
 - mouseover, 274
 - scroll, 280
 - zderzenia, 156
- zegary, 279
- zmienna, 44
 - ANCESTRY_FILE, 112
 - arguments, 97
 - document, 244
 - innerHeight, 276
 - this, 125, 151
- zmienne
 - globalne, 65
 - lokalne, 65
- zmiennosć, 88
- zmniejszanie ilości śmieci, 439
- znacznik, 238
 - <body>, 239
 - <canvas>, 330
 - <form>, 354
 - <head>, 239
 - <html>, 239, 244
 - , 255, 319

- <input>, 354
- <pre>, 252
- <script>, 240, 277
- <select>, 355
- , 252
- <style>, 257
- <table>, 293
- <textarea>, 355
- <title>, 239
 - otwierający, 238
 - zamykający, 238
- znajdowanie elementów, 248
- znak, 183
 - \$, 188
 - &, 239
 - ^, 188
- znaki
 - międzynarodowe, 200
 - nowego wiersza, 35
- zwracanie wartości, 48

Ż

- żądania
 - asynchroniczne, 341
 - HTTP, 339, 349
 - idempotentne, 401
- żądanie, 336
 - DELETE, 336, 400
 - GET, 336, 340
 - POST, 336, 343
 - PUT, 336, 348

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**

POZNAJ MOŻLIWOŚCI JĘZYKA JAVASCRIPT!

Współczesne aplikacje i strony internetowe nie byłyby takie, jakie są obecnie, gdyby nie JavaScript. Ten język programowania jeszcze kilka lat temu przeżywał kryzys, lecz w końcu został doceniony. Obecnie jest stosowany wszędzie tam, gdzie użytkownicy wymagają najwyższej interaktywności, szybkości działania oraz wygody korzystania z aplikacji internetowej. Jeżeli chcesz poznać JavaScript i użyć go już w najbliższym projekcie, to trafiłeś na doskonałą książkę.

Otwórz ją i przekonaj się, jak wygląda składnia JavaScriptu oraz typowe konstrukcje w tym języku. W trakcie lektury kolejnych rozdziałów nauczysz się budować przejrzystą strukturę programu, korzystać z obiektów i tablic oraz wyrażeń regularnych. Ponadto poznasz tajniki programowania obiektowego i najlepsze techniki obsługi błędów. Gdy opanujesz już podstawy związane z językiem, przyjdzie czas na drugą część książki, poświęconą możliwościom JavaScriptu w środowisku przeglądarki. Na

kolejnych stronach znajdziesz informacje o modelu DOM, korzystaniu z elementu canvas oraz obsłudze formularzy. Na koniec poznasz tajniki optymalizacji kodu, żeby w jeszcze większym stopniu wykorzystała potencjał JavaScriptu. Książka ta jest doskonałą lekturą dla czytelników, którzy chcą bezproblemowo wkroczyć w świat tego języka!

Dzięki tej książce:

- poznasz składnię języka JavaScript
- wykorzystasz jego konstrukcje
- poznasz techniki programowania obiektowego
- zmodyfikujesz model DOM strony WWW
- w pełni wykorzystasz potencjał JavaScriptu

Marijn Haverbeke — niezależny programista i pisarz. W centrum jego zainteresowań znajdują się języki programowania oraz narzędzia dla programistów. Jest autorem edytora CodeMirror, a także narzędzi Tern oraz Acorn.

Helion 	
33823	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/nowości>



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

ISBN 978-83-283-0969-2



9 788328 309692

cena: 79,00 zł



KOD KORZYSCI