JavaScript dla każdego!

Rusz głową!

Programowanie w JavaScript

Nauka języka JavaScript jeszcze nigdy nie była tak przyjemna!

Uważaj na często spotykane pułapki i niebezpieczeństwa

Unikaj żenujących błędów związanych z konwersjami typów

Spróbuj rozwikłać ponad 120 zagadek i ćwiczeń

Przeczytaj choć jeden rozdział, by przyspieszyć rozwój swojej kariery

Dowiedz się, dlaczego wszystko, co Twoi znajomi wiedzą o funkcjach i obiektach, najprawdopodobniej jest jedną wielką pomyłką

O'REILLY®

Eric T. Freeman, Elisabeth Robson

Helion

Tytuł oryginału: Head First JavaScript Programming

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-9880-6

© 2015 Helion S.A.

Authorized Polish translation of the English edition of Head First JavaScript Programming
9781449340131 © 2014 Eric Freeman and Elisabeth Robson.

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

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

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

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

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

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści (skrócony)

	Wprowadzenie	25
1.	Szybki skok na głębokie wody JavaScriptu: *Czas się zamoczyć*	39
2.	Pisanie prawdziwego kodu: *Idziemy dalej*	81
3.	Przedstawienie funkcji: *Stawiamy na funkcjonalność*	115
4.	Porządkowanie naszych danych: *Tablice*	159
5.	Zrozumieć obiekty: *Wycieczka do Obiektowa*	205
6.	Interakcja ze stronami WWW: *Poznajemy DOM*	259
7.	Typy, równość, konwersje i cały ten jazz: *Poważne typy*	293
8.	Łączenie wszystkiego w całość: *Tworzenie aplikacji*	343
9.	Programowanie asynchroniczne: *Obsługa zdarzeń*	407
10.	Funkcje pierwszej klasy: *Wyzwolone funkcje*	453
11.	Funkcje anonimowe, zasięg i domknięcia: *Poważne funkcje*	499
12.	Zaawansowane sposoby konstruowania obiektów: *Tworzenie obiektów*	543
13.	Stosowanie prototypów: *Obiekty ekstramocne*	583
A	Dziesięć najważniejszych zagadnień (których nie opisaliśmy): *Pozostałości*	643

Spis treści (pełny)

W

Wprowadzenie

Twój mózg koncentruje się na języku JavaScript. W tym miejscu *Ty* usiłujesz się czegoś *nauczyć*, a Twój *mózg* robi Ci przysługę i stara się, by wszystkie poznane informacje zostały *zapomniane*. Twój mózg myśli sobie: „Lepiej zostawić miejsce na naprawdę ważne informacje, takie jak dzikie zwierzęta, których należy unikać, albo czy jeżdżenie nago na snowboardzie jest dobrym pomysłem". W jaki sposób *możesz* przekonać swój mózg, by uznał, że Twoje życie zależy od znajomości JavaScriptu?

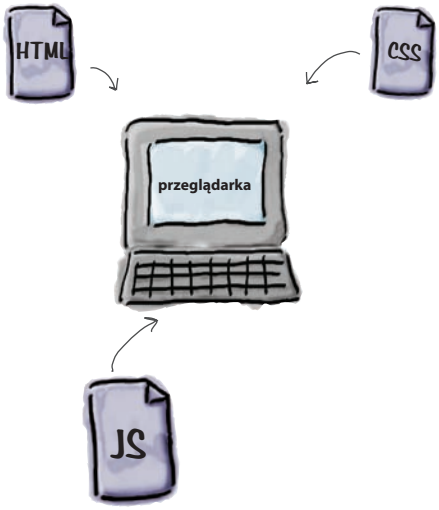
Dla kogo jest ta książka?	26
Wiemy, co myślisz	27
Wyobrażamy sobie, że czytelnik tej książki jest uczniem	28
Metapoznanie — myślenie o myśleniu	29
To, co MY zrobiliśmy	30
To, co TY możesz zrobić, aby zmusić swój mózg do posłuszeństwa	31
Przeczytaj to	32
Recenzenci techniczni	35
Podziękowania*	36

Kup książkę Poleć książkę

1 Szybki skok na głębokie wody JavaScriptu

Czas się zamoczyć

JavaScript to dane Ci supermoce. To **prawdziwy język programowania** internetu, który pozwala **dodawać** do stron WWW **zachowania**. Możesz zapomnieć o suchych, nudnych i statycznych stronach — z pomocą języka JavaScript będziesz w stanie porozumieć się ze swoimi użytkownikami, pobierać z internetu dane do wyświetlania na swoich stronach, rysować grafikę bezpośrednio na stronach i robić wiele innych, świetnych rzeczy. Co więcej, znając JavaScript, będziesz także mógł zapewniać swoim użytkownikom **całkowicie nowe** możliwości.



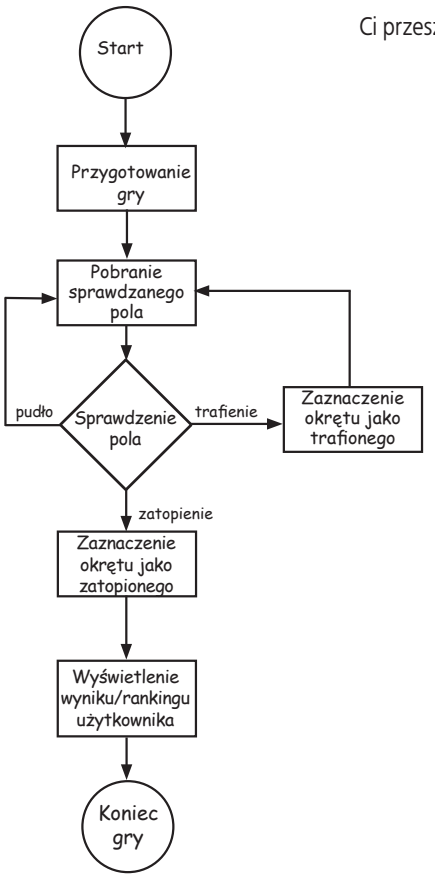
Sposób działania języka JavaScript	40
Jak należy pisać kod JavaScript?	41
Jak umieszczać kod JavaScript na stronie?	42
Dziecinko, JavaScript przebył długą drogę…	44
Jak tworzyć instrukcje?	48
Zmienne i wartości	49
Odsuń się od tej klawiatury!	50
Wyrazić się	53
Wykonywanie operacji więcej niż jeden raz	55
Jak działa pętla while?	56
Podejmowanie decyzji w języku JavaScript	60
A kiedy trzeba podejmować WIELE decyzji…	61
Wyciągnij rękę i nawiąż kontakt z użytkownikami	63
Poznajemy bliżej funkcję console.log	65
Otwieranie konsoli	66
Piszemy poważną aplikację JavaScript	67
Jak mogę dodać kod do strony? (Niech policzę wszystkie sposoby)	70
Będziemy musieli was rozdzielić	71

Kup książkę Poleć książkę

2 Pisanie prawdziwego kodu

Idziemy dalej

Już znasz zmienne, typy, wyrażenia…, możesz zatem pójść krok dalej. Chodzi o to, że już trochę poznałeś język JavaScript. Wiesz na tyle dużo, by napisać jakiś **prawdziwy kod**. Kod, który robi coś interesującego, którego ktoś chciałby używać. Jednak wciąż brakuje Ci **praktycznego doświadczenia** w pisaniu kodu. Właśnie mamy zamiar temu zaradzić, tu i teraz! A w jaki sposób? Skacząc na główkę na głęboką wodę i pisząc prostą grę, w całości w języku JavaScript. Cel jest bardzo ambitny, jednak będziemy go realizować krok po kroku. Chodź, zaczynamy! Jeśli zechcesz przy okazji uruchomić kolejny prosty startup, nie będziemy Ci przeszkadzać — kod jest Twój.



No i proszę. Prawdziwy schemat blokowy.

Napiszmy grę w okręty	82
Pierwsza próba…	82
Punkt pierwszy: projekt wysokiego poziomu	83
Analiza pseudokodu	85
A… zanim przejdziemy dalej, nie zapomnij o kodzie HTML	87
Pisanie kodu prostej wersji gry w okręty	88
A teraz zajmijmy się logiką gry	89
Krok pierwszy: przygotowanie pętli i pobranie danych	90
Jak działa funkcja prompt?	91
Sprawdzanie komórki wskazanej przez użytkownika	92
Czy użytkownikowi udało się trafić?	94
Dodanie kodu wykrywającego trafienia	95
Prezentacja informacji o zakończonej grze	96
To koniec implementacji logiki	98
Chwilka na zapewnianie jakości	99
Czy możemy pogadać o rozwlekłości Twojego kodu?	103
Kończymy prostą wersję gry w okręty	104
Jak przypisywać wartości losowe?	105
Najlepszy na świecie przepis na generowanie liczb losowych	105
Wróćmy do zapewniania jakości	107
Gratulujemy pierwszego prawdziwego programu w języku JavaScript i mamy dwa słowa o wielokrotnym używaniu kodu	109

Kup książkę Poleć książkę

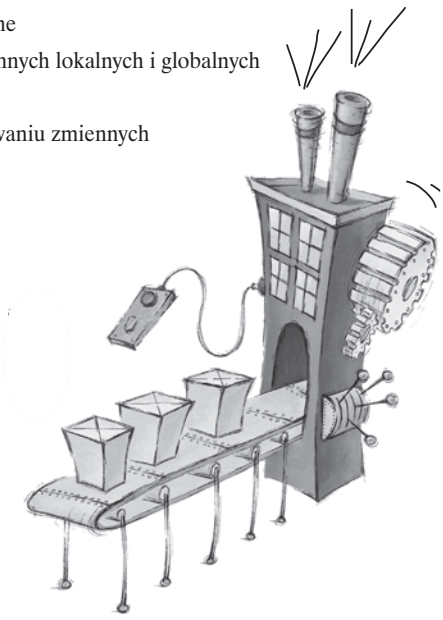
3

Przedstawienie funkcji

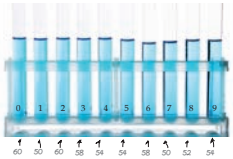
Stawiamy na funkcjonalność

Przygotuj się na użycie pierwszej ze swoich supermocy. Zdobyłeś już nieco umiejętności programistycznych; teraz nadszedł czas, aby rozwinąć je jeszcze bardziej przy użyciu **funkcji**. Funkcje zapewniają możliwość pisania kodu, który można stosować we wszelkich możliwych okolicznościach, kodu **używanego wielokrotnie**, którym można znacznie łatwiej zarządzać i w końcu który można **wyodrębnić**, nadać mu łatwą do zapamiętania nazwę, zapomnieć o całej jego złożoności i zająć się innymi ważnymi problemami. Przekonasz się, że funkcje to nie tylko droga, która zmieni Cię z autora skryptów w programistę. Są one kluczowym czynnikiem określającym styl programowania w języku JavaScript. W tym rozdziale zaczniemy od podstaw: poznasz mechanikę funkcji i tajniki ich działania, a dalej w tej książce będziesz stopniowo powiększać swoją wiedzę i umiejętności ich stosowania. A zatem, zacznij budować solidne podstawy znajomości JavaScriptu i zrób to *już teraz*.

Co z tym kodem było nie tak? 117
Swoją drogą, czy wspominaliśmy już o FUNKCJACH? 119
No dobrze, ale jak to właściwie działa? 120
Co można przekazywać do funkcji? 125
JavaScript przekazuje przez wartość 128
Zakręcone funkcje 130
Funkcje mogą także coś zwracać 131
Śledzenie wykonania funkcji z instrukcją return 132
Zmienne globalne i lokalne 135
Poznawanie zasięgu zmiennych lokalnych i globalnych 137
Krótkie życie zmiennych 138
Nie zapominaj o deklarowaniu zmiennych 139



Kup książkę Poleć książkę



4 Porządkowanie naszych danych
Tablice

JavaScript to nie tylko liczby, łańcuchy znaków i wartości logiczne. Dotychczas pisałeś jedynie kod JavaScript, w którym były używane wartości **typów prostych** — proste łańcuchy znaków, liczby i wartości logiczne, takie jak `„Burek"`, `23` oraz `true`. Korzystając z takich wartości, można zrobić naprawdę dużo, jednak w którymś momencie będziesz musiał zacząć posługiwać się znacznie **większą ilością danych**. Przykładowo mogą to być wszystkie produkty umieszczone w koszyku zakupowym albo utwory na liście odtwarzania, albo gwiazdozbiory i współrzędne poszczególnych gwiazd, albo cały katalog produktów. Jednak do tego potrzebujesz czegoś bardziej... *sexy*. W języku JavaScript preferowanym typem danych dla takich uporządkowanych zbiorów informacji jest **tablica**, a w tym rozdziale dokładnie przeanalizujemy, jak umieszczać dane w tablicach, przekazywać tablice oraz jak na nich operować. Dalej w książce omówimy także kilka innych sposobów **strukturyzacji danych**, jednak zaczniemy od tablic.

Czy możesz pomóc firmie BańkoCorp? 160
Jak reprezentować wiele wartości w JavaScripcie? 161
Jak działają tablice? 162
A w ogóle jak duża jest tablica? 164
Korpo-zdanio-budowator 166
W międzyczasie w firmie BańkoCorp… 169
Jak pobrać wszystkie elementy tablicy? 172
Chwila, istnieje lepszy sposób iteracji po tablicy 174
Czy możemy porozmawiać o rozwlekłości Twojego kodu? 180
Poprawienie pętli for przy użyciu operatora postinkrementacji 181
Szybka jazda próbna 181
Tworzenie pustej tablicy (i dodawanie do niej danych) 185
Zwycięzcami są… 189
Krótka inspekcja kodu… 191
Piszemy funkcję printAndGetHighScore 192
Refaktoryzacja kodu z użyciem funkcji printAndGetHighScore 193
Zastosowanie zmian… 195

Kup książkę Poleć książkę

5 Zrozumieć obiekty

Wycieczka do Obiektowa

Do tej pory w tworzonym kodzie używałeś jedynie danych typów prostych oraz tablic. Dodatkowo podchodziłeś do programowania w **sposób proceduralny** — korzystałeś z prostych instrukcji, instrukcji warunkowych oraz pętli, ewentualnie umieszczałeś je w funkcjach — to właściwie nie jest **programowanie obiektowe**. Prawdę powiedziawszy, to *w ogóle* nie jest programowanie obiektowe! Tu i tam, nawet o tym nie wiedząc, użyłeś — co prawda — kilku obiektów, jednak na razie jeszcze nie napisałeś żadnego własnego obiektu. Nadszedł najwyższy czas, żeby zostawić to stare i nudne proceduralne miasteczko i zacząć tworzenie własnych **obiektów**. W tym rozdziale dowiesz się, dlaczego stosowanie obiektów sprawi, że Twoje życie stanie się znacznie lepsze — przynajmniej pod **względem programistycznym** (niestety, w jednej książce nie możemy poprawić Twojej znajomości mody *i jednocześnie* nauczyć programowania w języku JavaScript). I jeszcze jedno ostrzeżenie: kiedy już poznasz obiekty, nigdy nie będziesz chciał ich porzucić. Wyślij nam pocztówkę, kiedy już dojedziesz do krainy obiektów.

Czy ktoś powiedział „obiekty”? 206
Myśląc o właściwościach… 207
W jaki sposób tworzy się obiekty? 209
Czym w ogóle jest programowanie obiektowe? 212
Jak działają właściwości? 213
W jaki sposób zmienna przechowuje obiekt? Ciekawe umysły chciałyby to wiedzieć… 218
Porównanie danych typów prostych i obiektów 219
Jeszcze inne operacje z wykorzystaniem obiektów… 220
Analiza działania wstępnej selekcji 222
Porozmawiajmy nieco więcej o przekazywaniu obiektów do funkcji 224
Zachowuj się! Jak dodawać zachowania do obiektów? 230
Poprawianie metody drive 231
Dlaczego metoda drive nic nie wie o właściwości started? 234
Jak działa this? 236
Jak zachowanie wpływa na stan? Dodajemy trochę paliwa 242
A teraz niech stan będzie mieć wpływ na zachowanie 243
Gratulujemy utworzenia pierwszych obiektów! 245
Wiesz co? Obiekty są wszędzie dookoła Ciebie (i ułatwiają Ci życie) 246

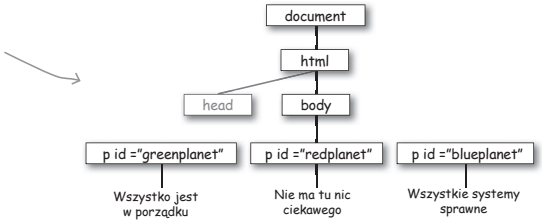
Kup książkę Poleć książkę

6 Interakcja ze stronami WWW

Poznajemy DOM

Przebyłeś już długą drogę, poznając JavaScript. Powoli z żółtodzioba zmieniłeś się w twórcę prostych skryptów, a potem w końcu w **programistę**. Jednak wciąż Ci czegoś brakuje. Abyś mógł naprawdę wykorzystać całą swoją znajomość języka JavaScript, musisz dowiedzieć się, jak prowadzić interakcję ze stronami WWW, w których umieszczasz swoje skrypty. Tylko to pozwoli Ci tworzyć strony, które są **dynamiczne**, które reagują, odpowiadają i aktualizują swoją zawartość już po jej wczytaniu przez przeglądarkę. W jaki sposób można prowadzić interakcję ze stroną WWW? Służy do tego **DOM**, nazywany także **obiektowym modelem dokumentu**. W tym rozdziale opiszemy go szczegółowo i wyjaśnimy, jak można z niego korzystać i jak go używać wraz z językiem JavaScript, by nauczyć nasze strony wykonywania wielu nowych sztuczek.

W poprzednim rozdziale miałeś wykonać niewielką misję — misję złamania kodu 260
Co robi ten kod? 261
Jak naprawdę wygląda interakcja JavaScriptu ze stroną WWW? 263
Jak wypiec swój własny DOM? 264
Pierwszy smak DOM 265
Pobieranie elementu przy użyciu metody getElementById 270
Co pobieramy z DOM? 271
Dostęp do kodu HTML w elemencie 272
Co się dzieje, kiedy zmieniamy DOM? 274
Jazda próbna wokół planet 277
Nawet nie myśl o uruchamianiu mojego kodu, zanim strona nie zostanie w całości wczytana 279
Ty mówisz: „przeglądarka”, ja mówię: „wywołanie zwrotne” 280
Jak ustawiać atrybuty przy użyciu metody setAttribute? 285
Więcej zabawy z atrybutami (wartości atrybutów można także POBIERAĆ) 286
Nie zapominaj, że także metoda getElementById może zwracać null 286
Tymczasem w systemie słonecznym… 287

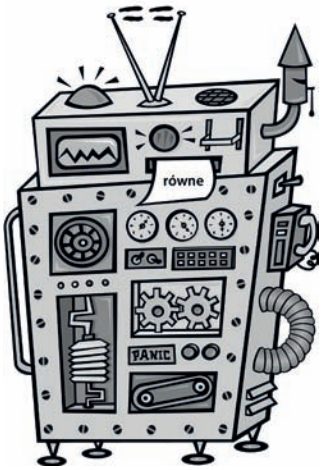


Kup książkę Poleć książkę

7 Typy, równość, konwersje i cały ten jazz

Poważne typy

Nadszedł czas, by poważnie przyjrzeć się typom. Jedną ze wspaniałych cech JavaScriptu jest to, że można w nim zrobić całkiem dużo bez jego szczegółowej znajomości. Aby jednak perfekcyjnie opanować język, dostać awans i zacząć robić to, co naprawdę chcesz robić w życiu, musisz się zaprzyjaźnić z typami. Czy pamiętasz, co już dawno, na samym początku książki powiedzieliśmy na temat JavaScriptu? Że nie może się poszczycić rozpuszczoną, popularną, akademicką definicją? No cóż... To prawda, jednak życie akademickie nie zatrzymało ani Steve'a Jobsa ani Billa Gatesa, nie zatrzymało także języka JavaScript. Oznacza to jednak, że JavaScript nie ma... hm... doskonale przemyślanego systemu typów i znajdziemy w nim sporo dziwactw. Nie obawiaj się jednak — w tym rozdziale dokładnie wszystko wyjaśnimy, dzięki czemu już niebawem nauczysz się unikać tych wszystkich zawstydzających problemów z typami.



Gdzieś tam jest ukryta prawda…	294
Uważaj, możesz natknąć się na undefined, kiedy będziesz się tego najmniej spodziewać…	296
Jak używać null?	299
Stosowanie wartości NaN	301
Sprawy stają się jeszcze dziwniejsze	301
Musimy coś wyznać	303
Zrozumienie operatora równości (pseudonim: ==)	304
Jak wykonywana jest konwersja operandów operatora równości (brzmi groźniej, niż jest w rzeczywistości)?	305
Jak ściśle podejść do zagadnienia równości?	308
Jeszcze więcej konwersji typów…	314
Jak określić, czy dwa obiekty są równe?	317
Gdzieś tam leży prawda…	319
JavaScript uwzględnia fałsz	320
Sekretne życie łańcuchów znaków	322
Dlaczego łańcuch może wyglądać jak dana typu prostego oraz jak obiekt?	323
Pięciominutowa wycieczka po metodach (i właściwościach) łańcuchów znaków	325
Wojna o fotel	329

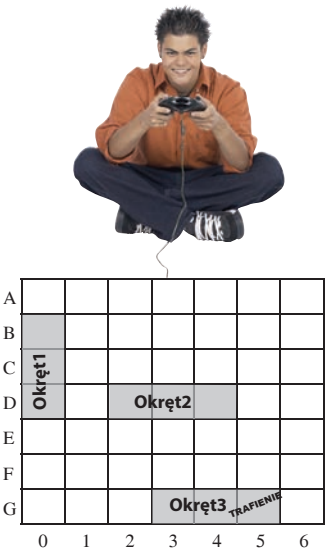
Kup książkę Poleć książkę

8

Łączenie wszystkiego w całość

Tworzenie aplikacji

Włóż to do swojego przybornika z narzędziami, czyli do skrzynki, w której umieszczasz wszystkie swoje nowe umiejętności programistyczne, wiedzę dotyczącą DOM, a nawet HTML i CSS. W tym rozdziale wykorzystasz wszystkie te umiejętności, by utworzyć pierwszą prawdziwą **aplikację internetową**. Wystarczy już **prymitywnych namiastek gier**, w których na jednowymiarowej planszy pływa jeden okręt. W tym rozdziale wykonasz **pełne doświadczenie**: dużą, atrakcyjną planszę do gry, wiele okrętów oraz obsługę wprowadzania danych przez użytkownika. Struktura strony, na której będzie prowadzona gra, powstanie w języku HTML, jej wygląd określony zostanie przy użyciu stylów CSS, a zachowanie samej gry napisane w języku JavaScript. Przygotuj się: to będzie jazda na całego, rozdział, w którym pełnym gazem będziesz zmierzać w kierunku pisania naprawdę poważnego kodu.



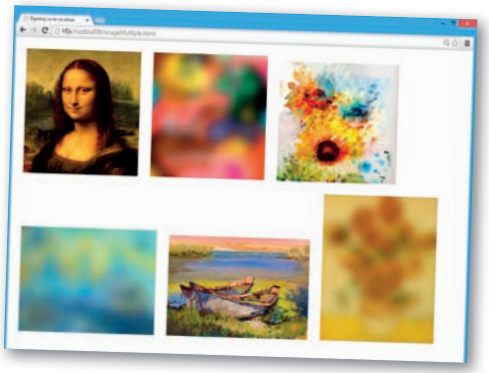
Tym razem napiszemy PRAWDZIWĄ grę w okręty 344
Krok wstecz… do HTML i CSS 345
Tworzenie strony HTML: postać ogólna 346
Dodawanie stylów 350
Stosowanie klas hit i miss 353
Jak zaprojektować grę? 355
Implementacja widoku 357
Jak działa metoda displayMessage? 357
Jak działają metody displayHit oraz displayMiss? 359
Model 362
W jaki sposób będziemy reprezentować okręty? 364
Implementacja obiektu modelu 367
Rozmyślamy o metodzie fire 368
Implementacja kontrolera 375
Przetwarzanie pola wskazanego przez użytkownika 376
Planowanie kodu… 377
Implementacja funkcji parseGuess 378
W międzyczasie w kontrolerze… 381
Dodanie procedury obsługi zdarzeń do przycisku Ognia! 385
Przekazywanie współrzędnych do kontrolera 386
Jak rozmieszczać okręty? 390
Implementacja metody generateShip 391
Generacja początkowego pola okrętu 392
Dokończenie metody generateShip 393

Kup książkę Poleć książkę

9

Programowanie asynchroniczne
Obsługa zdarzeń

Po przeczytaniu tego rozdziału zdasz sobie sprawę z tego, że to już nie są przelewki i nie jesteśmy już w Kansas. Do tej pory pisałeś kod, który zazwyczaj wykonywany był od samego początku do końca — oczywiście Twój kod mógł być nieco bardziej skomplikowany i zawierać kilka funkcji, obiektów i metod, jednak w jakimś momencie ten kod po prostu był wykonany wiersz po wierszu. Cóż, jest nam bardzo przykro, że mówimy Ci to tak późno, jednak **typowy kod JavaScript nie jest pisany w taki sposób**. Kod pisany w tym języku **reaguje na zdarzenia**. Jakiego rodzaju zdarzenia? Może to być kliknięcie strony przez użytkownika, przesłanie danych z serwera, upływ pewnego okresu czasu w przeglądarce, jakaś zmiana wprowadzona w DOM oraz wiele innych. W rzeczywistości, w niewidoczny dla nas sposób, w przeglądarce **cały czas** zachodzą jakieś zdarzenia. W tym rozdziale jeszcze raz przemyślisz swoje podejście do sposobu pisania kodu JavaScript i dowiesz się, dlaczego trzeba pisać kod reagujący na zdarzenia oraz jak należy to robić.



Czym są zdarzenia? 409
Czym jest procedura obsługi zdarzeń? 410
Jak napisać pierwszą procedurę obsługi zdarzeń? 411
Jazda próbna procedury obsługi zdarzeń 412
Poznajemy zdarzenia, pisząc grę 414
Implementacja gry 415
Jazda próbna 416
Dodajmy więcej obrazków 420
Teraz musimy przypisać tę samą procedurę obsługi zdarzeń do właściwości onclick każdego obrazka 421
Jak użyć tej samej funkcji do obsługi wszystkich obrazków? 422
Jak działa obiekt zdarzenia? 425
Zaprzęgamy obiekt zdarzenia do pracy 427
Testujemy obiekt zdarzenia i właściwość target 428
Zdarzenia i kolejki 430
Jeszcze więcej zdarzeń 433
Jak działa funkcja setTimeout? 434
Kończenie gry 438
Jazda testowa z licznikiem czasu 439

Kup książkę Poleć książkę

10

Funkcje pierwszej klasy

Wyzwolone funkcje

Poznaj funkcje, a potem baw się na całego. Każda sztuka, rzemiosło czy też dyscyplina sportowa mają swoją kluczową cechę, która odróżnia graczy przeciętnych od wirtuozów. W przypadku języka JavaScript jest to prawdziwe i dokładne zrozumienie funkcji. W języku JavaScript funkcje mają kluczowe znaczenie, a wiele technik służących do **projektowania i organizacji** kodu bazuje na zaawansowanej znajomości funkcji i umiejętności korzystania z nich. Droga prowadząca do poznania funkcji na takim poziomie jest interesująca i niejednokrotnie wymagająca dla mózgu, zatem dobrze się do niej przygotuj. W tym rozdziale będziesz się czuł jak dziecko oprowadzane przez pana Willy'ego Wonkę po fabryce czekolady — będziesz w nim kontynuował poznawanie funkcji w języku JavaScript, a przy okazji zobaczysz dziwaczne, wariackie i cudowne rzeczy.

Tajemnicze, podwójne życie słowa kluczowego function	454
Deklaracje funkcji a wyrażenia funkcyjne	455
Przetwarzanie deklaracji funkcji	456
I co dalej? Przeglądarka wykonuje kod	457
Idziemy dalej… Instrukcja warunkowa	458
O tym, dlaczego funkcje są także wartościami	463
Czy wspominaliśmy już, że w JavaScripcie funkcje mają status „pierwszej klasy”?	466
Latanie pierwszą klasą	467
Piszemy kod do przetwarzania i sprawdzania pasażerów	468
Przetwarzanie listy pasażerów	470
Przekazywanie funkcji do funkcji	471
Zwracanie funkcji z funkcji	474
Pisanie kodu do wydawania napojów	475
Pisanie kodu do wydawania napojów — inne podejście	476
Przyjmowanie zamówień z wykorzystaniem funkcji pierwszej klasy	478
Cola sieciowicka	481
Jak działa metoda sort tablic?	483
Łączymy wszystko w całość	484
Weźmy teraz sortowanie na jazdę próbną	486

Kup książkę Poleć książkę

11

Funkcje anonimowe, zasięg i domknięcia

Poważne funkcje

W poprzednim rozdziale rozłożyłeś funkcje na czynniki pierwsze, ale wciąż musisz się o nich jeszcze sporo dowiedzieć. W tym rozdziale będzie prawdziwa jazda na całego. Pokażemy Ci, jak **naprawdę korzysta się** z funkcji. To nieszczególnie długi rozdział, jednak bardzo intensywny, a po jego przeczytaniu siła wyrazu tworzonego przez Ciebie kodu JavaScript będzie większa, niż mógłbyś przypuszczać. Co więcej, mamy w nim zamiar przedstawić pewne ogólnie przyjęte idiomy i konwencje związane z tworzeniem i stosowaniem funkcji w języku JavaScript, dzięki czemu będziesz już mógł skorzystać z kodu pisanego przez współpracowników lub czerpanego z ogólnie dostępnych bibliotek JavaScript. A jeśli jeszcze nigdy nie słyszałeś o **funkcjach anonimowych** i **domknięciach** (ang. *closure*), to wiedz, że znalazłeś się w odpowiednim miejscu.

Rzut oka na inną stronę funkcji… 500
Jak używać funkcji anonimowych? 501
Musimy ponownie pomówić o rozwlekłości Twojego kodu 503
Kiedy funkcja zostaje zdefiniowana? To zależy… 507
Co się właśnie stało? Dlaczego funkcja fly nie była zdefiniowana? 508
Zagnieżdżanie funkcji 509
Jaki wpływ na zasięg ma zagnieżdżanie funkcji? 510
Krótka powtórka z zasięgu leksykalnego 512
Miejsce, w którym zasięg leksykalny sprawia, że sprawy stają się interesujące 513
Funkcje raz jeszcze 515
Wywoływanie funkcji (po raz wtóry) 516
Czym właściwie są domknięcia? 519
Domykanie funkcji 520
Zastosowanie domknięć w celu zaimplementowania magicznego licznika 522
Zaglądamy za kulisy… 523
Tworzenie domknięcia poprzez przekazanie wyrażenia funkcyjnego jako argumentu 525
Domknięcia zawierają rzeczywiste środowisko, a nie jego kopię 526
Tworzenie domknięć jako procedur obsługi zdarzeń 527
Jak działa domknięcie liczące kliknięcia? 530

Kup książkę Poleć książkę

12

Zaawansowane sposoby konstruowania obiektów

Tworzenie obiektów

Dotychczas wszystkie obiekty tworzyłeś własnoręcznie. Opracowując każdy z nich, korzystałeś z **literału obiektowego**, w którym podawałeś wszystkie właściwości i metody. Na niewielką skalę takie rozwiązanie będzie się sprawdzać, jednak podczas tworzenia poważnego kodu będziesz potrzebował czegoś lepszego. Właśnie w tym miejscu do akcji wkraczają **konstruktory obiektów**. Konstruktory sprawiają, że tworzenie obiektów jest znacznie łatwiejsze, a wszystkie budowane obiekty mogą być zgodne z jednym **wzorcem** — oznacza to, że konstruktorów używamy po to, by zapewnić, że wszystkie obiekty będą miały te same właściwości i udostępniały te same metody. Kiedy korzystasz z konstruktorów, kod obiektów może być znacznie bardziej **zwięzły** i mniej podatny na występowanie błędów, zwłaszcza w przypadkach, gdy tworzysz bardzo dużo obiektów. Po przeczytaniu tego rozdziału będziesz stosował konstruktory z taką wprawą, jakbyś dorastał w Obiektowie.

Tworzenie obiektów przy użyciu literałów obiektowych 544
Stosowanie konwencji podczas tworzenia obiektów 545
Prezentacja konstruktorów obiektów 547
Jak utworzyć konstruktor? 548
Jak należy używać konstruktorów? 549
Sposób działania konstruktorów 550
W konstruktorach można także umieszczać metody 552
Nadszedł czas na produkcję masową 558
Weźmy nowe samochody na jazdę próbną 560
Nie zapominaj jeszcze o literałach obiektowych 561
Przekazywanie argumentów przy użyciu literału obiektowego 562
Modyfikacja konstruktora Car 563
Zrozumieć instancje obiektów 565
Nawet obiekty utworzone przy użyciu konstruktora mogą mieć własne właściwości 568
Konstruktory stosowane w praktyce 570
Obiekt Array 571
Jeszcze więcej zabawy z wbudowanymi obiektami JavaScriptu 573

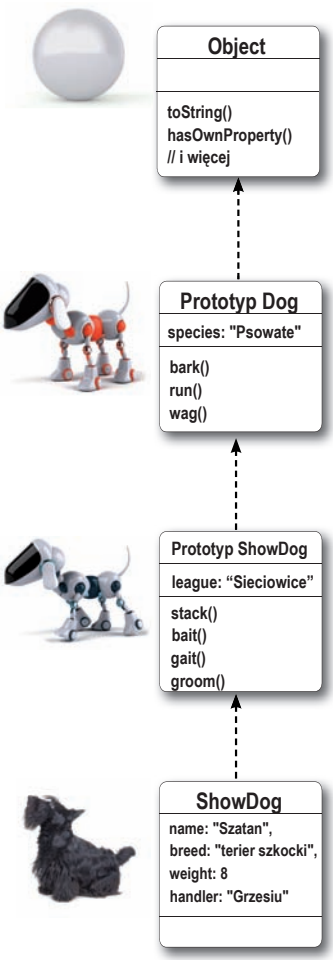
Kup książkę Poleć książkę

13

Stosowanie prototypów

Obiekty ekstramocne

Nauka tworzenia obiektów była jedynie początkiem. Nadszedł czas na wzmocnienie obiektów. Potrzebujesz więcej sposobów, by tworzyć wzajemne **związki** pomiędzy obiektami oraz zapewniać możliwość **współdzielenia kodu** przez takie powiązane obiekty. Dodatkowo potrzebujesz także sposobów na rozszerzanie i zwiększanie możliwości istniejących obiektów. Innymi słowy, potrzebujesz więcej narzędzi. W tym rozdziale przekonasz się, że JavaScript dysponuje naprawdę użytecznym **modelem obiektowym**, choć jednocześnie jest on nieco odmienny od modeli stosowanych w innych obiektowych językach programowania. Zamiast typowego modelu obiektów bazującego na klasach, w JavaScripcie wykorzystano model bazujący na **prototypach**, w którym obiekty mogą dziedziczyć po innych obiektach i rozszerzać ich zachowania. A co on daje? Już wkrótce się przekonasz. A zatem zaczynajmy...



Hej, zanim zaczniemy, mamy lepszy sposób rysowania diagramów obiektów	585
Ponowna analiza konstruktorów: wielokrotnie używamy kodu, ale czy robimy to efektywnie?	586
Czy powielanie metod jest poważnym problemem?	588
Czym są prototypy?	589
Dziedziczenie po prototypie	590
Jak działa dziedziczenie?	591
Przesłanianie prototypu	593
Jak przygotować prototyp?	596
Prototypy są dynamiczne	602
Bardziej interesująca implementacja metody sit	604
Jeszcze jeden raz: sposób działania właściwości sitting	605
Jak zaprojektować psa pokazowego?	609
Tworzenie łańcucha prototypów	611
Jak działa dziedziczenie w łańcuchu prototypów?	612
Tworzenie prototypu psa pokazowego	614
Tworzenie instancji psa pokazowego	618
Analiza wyników ćwiczenia	621
Łańcuch prototypów nie kończy się na psie	627
Wykorzystanie dziedziczenia na swoją korzyść… Przesłonięcie domyślnych metod	628
Stosowanie dziedziczenia do własnych celów… Rozszerzanie wbudowanych obiektów	630
Wielka Jednolita Teoria JavaScriptu	632
Łączenie wszystkiego w całość	633
Co dalej?	633

Kup książkę Poleć książkę

A

Pozostałości

Dziesięć najważniejszych rzeczy (których nie opisaliśmy)

Opisaliśmy naprawdę sporo zagadnień i już niemal udało Ci się skończyć tę książkę. Będziemy za Tobą tęsknili, jednak zanim pozwolimy Ci odejść, musimy jeszcze coś powiedzieć, bo nie czulibyśmy się w porządku, wypuszczając Cię w świat bez tych informacji. Nie ma możliwości, byśmy w tym stosunkowo niewielkim rozdziale zdołali zmieścić wszystko, co ewentualnie mogłoby się przydać. Prawdę mówiąc, wcześniej *zamieściliśmy* w tym rozdziale wszystko to, co przydałoby się, żebyś wiedział o programowaniu w języku JavaScript, lecz musieliśmy zmniejszyć rozmiar czcionki do 0,00004 punktu. Wszystko się zmieściło, ale nikt nie był w stanie tego przeczytać. Dlatego większość tego tekstu odrzuciliśmy, pozostawiając tu jedynie Dziesięć Najważniejszych Zagadnień. I to naprawdę *jest koniec* tej książki. Oczywiście z wyjątkiem indeksu, który koniecznie musisz przeczytać!

1. jQuery 644
2. Więcej operacji na DOM 646
3. Obiekt window 647
4. Obiekt arguments 648
5. Obsługa wyjątków 649
6. Dodawanie procedur obsługi zdarzeń przy użyciu metody addEventListener 650
7. Wyrażenia regularne 652
8. Rekurencja 654
9. JSON 656
10. JavaScript po stronie serwera 657

S

Skorowidz 661

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Poważne funkcje

W poprzednim rozdziale rozłożyłeś funkcje na czynniki pierwsze, ale wciąż musisz się o nich jeszcze sporo dowiedzieć. W tym rozdziale będzie prawdziwa jazda na całego. Pokażemy Ci, jak **naprawdę korzysta się** z funkcji. To nieszczególnie długi rozdział, jednak bardzo intensywny, a po jego przeczytaniu siła wyrazu tworzonego przez Ciebie kodu JavaScript będzie większa, niż mógłbyś przypuszczać. Co więcej, mamy w nim zamiar przedstawić pewne ogólnie przyjęte idiomy i konwencje związane z tworzeniem i stosowaniem funkcji w języku JavaScript, dzięki czemu będziesz już mógł skorzystać z kodu pisanego przez współpracowników lub czerpanego z ogólnie dostępnych bibliotek JavaScript. A jeśli jeszcze nigdy nie słyszałeś o **funkcjach anonimowych** i **domknięciach** (ang. *closure*), to wiedz, że znalazłeś się w odpowiednim miejscu.

A może słyszałeś o domknięciach, lecz nie wiesz, co to jest? Tym bardziej jest to rozdział, który powinieneś przeczytać!

Kup książkę Poleć książkę

Rzut oka na inną stronę funkcji...

Poznałeś już dwie strony funkcji: tę formalną, deklaratywną stronę funkcji oraz znacznie bardziej ekspresyjną stronę wyrażeń funkcyjnych. A teraz nadszedł czas, by przedstawić ich jeszcze inną, interesującą stronę, czyli *funkcje anonimowe*.

Mówiąc o funkcjach anonimowych, mamy na myśli *funkcje, które nie mają nazwy*. Jak coś takiego jest możliwe? No cóż, kiedy tworzymy funkcje z wykorzystaniem deklaracji, *bez wątpienia mają one nazwy*. Jednak w przypadku, gdy budujemy funkcje przy użyciu wyrażeń funkcyjnych, *nie musimy podawać ich nazw*.

Pewnie sobie pomyślałeś, że to całkiem interesujące i zapewne naprawdę można tak zrobić, ale co z tego? Kiedy użyjemy funkcji anonimowych, niejednokrotnie możemy znacząco skrócić nasz kod, a także sprawić, że będzie bardziej zwięzły i czytelny, efektywniejszy i łatwiejszy w utrzymaniu.

A zatem, przekonajmy się, jak można używać funkcji anonimowych. Zaczniemy od fragmentu kodu, który poznałeś już wcześniej, i pokażemy, jak wykorzystywać funkcje anonimowe.

To jest procedura obsługi zdarzeń load, którą tworzymy w standardowy sposób.

Najpierw definiujemy funkcję. Funkcja ta ma nazwę handler.

```
function handler() { alert("O tak, strona została wczytana!"); }
window.onload = handler;
```

Następnie zapisujemy tę funkcję we właściwości onload obiektu window, używając jej nazwy handler.

A kiedy strona zostanie wczytana, przeglądarka wywoła funkcję handler.

Zaostrz ołówek

Skorzystaj ze swojej wiedzy na temat funkcji i zmiennych, aby wskazać, które z poniższych stwierdzeń są prawdziwe.

- ❑ Zmienna `handler` zawiera referencję do funkcji.
- ❑ Kiedy przypisujemy `handler` właściwości `window.onload`, zapisujemy w niej referencję do funkcji.
- ❑ Jedynym powodem istnienia zmiennej `handler` jest zapisanie jej we właściwości `window.onload`.
- ❑ Już nigdy więcej nie użyjemy funkcji `handler`, gdyż jest to kod, który z założenia ma być wykonywany wyłącznie podczas pierwszego wczytania strony.
- ❑ Dwukrotne wywoływanie procedury obsługi zdarzeń `load` nie jest dobrym pomysłem — może ono doprowadzić do wystąpienia problemów, gdyż ten kod służy zazwyczaj do wykonywania czynności związanych z inicjalizacją całej strony.
- ☐ Wyrażenia funkcyjne tworzą referencje do funkcji.
- ☐ Czy wspominaliśmy, że przypisując funkcję `handler` właściwości `window.onload`, zapisujemy w niej referencję do funkcji?

Kup książkę Poleć książkę

Jak używać funkcji anonimowych?

A zatem chcemy utworzyć funkcję do obsługi zdarzeń `load`, wiemy jednak, że jest to „funkcja jednorazowa”, gdyż zdarzenie to jest generowane tylko jeden raz podczas całego okresu prezentacji strony w przeglądarce. Możemy także zauważyć, że we właściwości `window.onload` jest zapisywana referencja do funkcji — a konkretnie rzecz biorąc, referencja do funkcji `handler`. Ponieważ jednak funkcja ta jest przeznaczona tylko do jednokrotnego użycia, zatem określanie jej nazwy jest niepotrzebne, gdyż używamy jej jedynie po to, by zapisać referencję we właściwości `window.onload`.

Zastosowanie funkcji anonimowej umożliwia oczyszczenie naszego kodu. Funkcja anonimowa jest po prostu wyrażeniem funkcyjnym, pozbawionym nazwy i zapisanym w miejscu, w którym normalnie użylibyśmy referencji do funkcji. Aby jednak to wszystko ze sobą powiązać, przeanalizujemy przykład wykorzystania wyrażenia funkcyjnego w sposób anonimowy.

```
function handler() { alert("O tak, strona została wczytana!"); }
window.onload = handler;
```

Najpierw usuwamy zmienną handler, tworząc tym samym wyrażenie funkcyjne.

```
function () { alert("O tak, strona została wczytana!"); }
window.onload =                                          ;
```

Następnie przypisujemy je bezpośrednio właściwości window.onload.

```
window.onload = function() { alert("O tak, strona została wczytana!"); }
```

Teraz procedura obsługi została przypisana bezpośrednio właściwości window.onload, bez konieczności stosowania niepotrzebnej nazwy.

Teraz kod jest znacznie bardziej zwarty. Niezbędną funkcję przypisujemy bezpośrednio właściwości onload. Oprócz tego, nie tworzymy nazwy funkcji, która przypadkowo mogłaby zostać użyta w innym miejscu kodu (w końcu nazwa „handler” jest stosowana dosyć często!).

Rany, popatrz! Nie ma nazwy!

WYSIL SZARE KOMÓRKI

Czy w którymś z przykładów przedstawionych wcześniej w tej książce były używane funkcje anonimowe, choć nie uprzedzaliśmy o tym?

Podpowiedź: może ukrywają się gdzieś w Twoich obiektach?

Kup książkę Poleć książkę

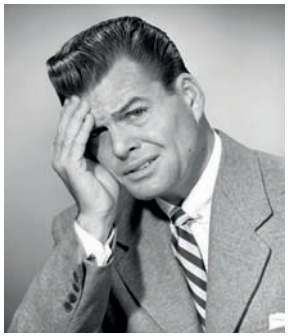
Zaostrz ołówek

Przedstawiony poniżej fragment kodu zapewnia kilka możliwości zastosowania funkcji anonimowych. Skorzystaj z nich i użyj funkcji anonimowych wszędzie tam, gdzie to możliwe. Możesz przekreślić stary kod i obok napisać nowy. I jeszcze jedna rzecz: zakreśl wszystkie anonimowe funkcje, które już są używane.

```
window.onload = init;
var cookies = {
    instructions: "Wstępne rozgrzewanie do 175 stopni...",
    bake: function(time) {
        console.log("Wypiekam ciasteczka.");
        setTimeout(done, time);
    }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Już można wypiekać ciasteczka.");
    cookies.bake(2500);
}
function done() {
    alert("Ciasteczka są gotowe, wyciągnij je, by przestygły.");
    console.log("Chłodzenie ciasteczek.");
    var cool = function() {
        alert("Ciasteczka są już zimne, można je jeść!");
    };
    setTimeout(cool, 1000);
}
```


Kup książkę Poleć książkę

Musimy ponownie pomówić o rozwlekłości Twojego kodu



Bardzo nam się nie podoba, że musimy wracać do tego zagadnienia, zwłaszcza że poświęciłeś dużo wysiłku na naukę funkcji — wiesz, jak je wywoływać, jak przypisywać zmiennym, jak przekazywać do innych funkcji i jak zwracać jako wynik wykonania innych funkcji — jednak wciąż pisany przez Ciebie kod jest bardziej rozwlekły, niż to konieczne (można by także powiedzieć, że nie jesteś tak ekspresyjny, jak mógłbyś być). Przyjrzyj się poniższemu przykładowi.

To zwyczajna funkcja o nazwie cookieAlarm, która wyświetla komunikat informujący, że ciasteczka są już gotowe.

```
function cookieAlarm() {
    alert("Już czas wyjąć ciasteczka z piekarnika.");
};

setTimeout(cookieAlarm, 600000);
```

Wygląda na to, że ciasteczka będą gotowe za 10 minut... tak tylko mówię.

Gdybyś zapomniał, to ten czas jest wyrażony w milisekundach, a zatem 1000 · 60 · 10 = 600 000.

A tu bierzemy tę funkcję i przekazujemy ją jako argument wywołania metody setTimeout.

Choć ten kod wygląda całkiem dobrze, jednak po zastosowaniu funkcji anonimowej możemy go nieco skrócić. W jaki sposób? No cóż… Pomyśl o zmiennej `cookieAlarm` umieszczonej w wywołaniu metody `setTimeout`. To zmienna, która odwołuje się do funkcji, a zatem w momencie wywoływania metody `setTimeout` jest do niej przekazywana referencja do funkcji. Już wiesz, że użycie zmiennej zawierającej referencję do funkcji jest jednym z kilku sposobów uzyskania takiej referencji, jednak — podobnie jak w przedstawionym kilka stron wcześniej przykładzie z właściwością `window.onload` — także i tu możemy skorzystać z funkcji anonimowej. Zmodyfikujmy zatem ten kod, używając w nim wyrażenia funkcyjnego.

Teraz zamiast zmiennej w wywołaniu metody setTimeout przekazujemy inną funkcję, zapisując jej kod bezpośrednio w wywołaniu.

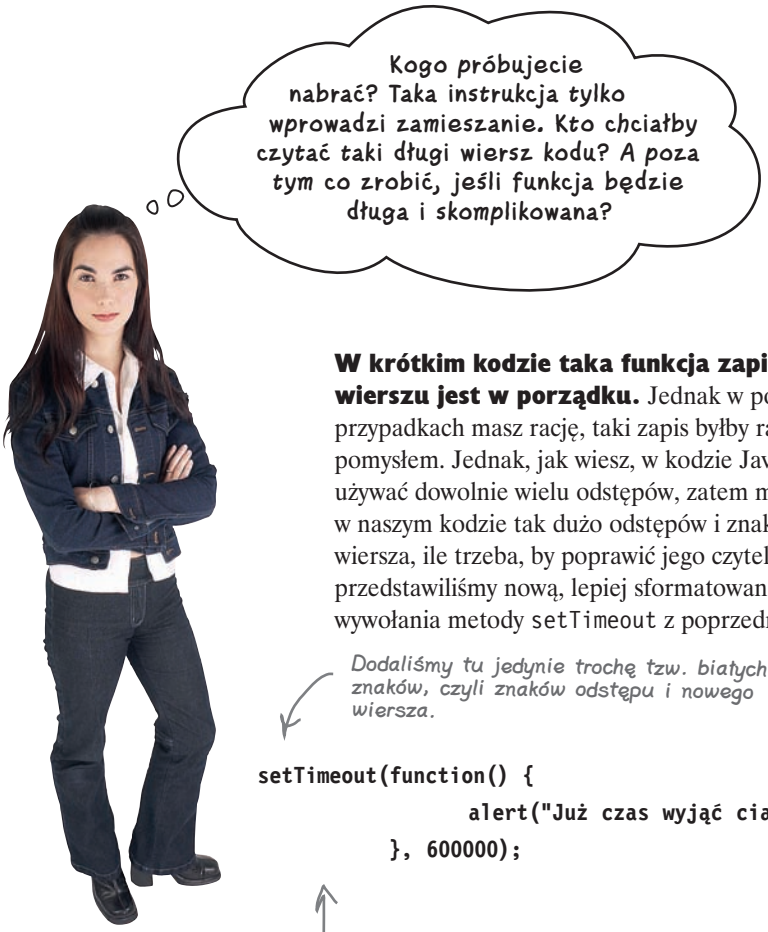
Zwróć szczególną uwagę na zastosowaną składnię. Użyliśmy całego wyrażenia funkcyjnego, kończącego się zamykającym nawiasem klamrowym, które zamknęliśmy przecinkiem umieszczonym przed kolejnym argumentem, dokładnie tak samo, jak robimy w przypadku wszystkich innych argumentów w wywołaniach funkcji.

```
setTimeout(function() { alert("Już czas wyjąć ciasteczka z piekarnika."); }, 600000);
```

Zapisaliśmy nazwę wywoływanej metody, setTimeout, za nią nawias otwierający i pierwszy argument wywołania — wyrażenie funkcyjne.

A tu za wyrażeniem funkcyjnym zapisaliśmy drugi argument.

Kup książkę Poleć książkę



W krótkim kodzie taka funkcja zapisana w jednym wierszu jest w porządku. Jednak w pozostałych przypadkach masz rację, taki zapis byłby raczej kiepskim pomysłem. Jednak, jak wiesz, w kodzie JavaScript można używać dowolnie wielu odstępów, zatem możemy umieścić w naszym kodzie tak dużo odstępów i znaków nowego wiersza, ile trzeba, by poprawić jego czytelność. Poniżej przedstawiliśmy nową, lepiej sformatowaną postać wywołania metody `setTimeout` z poprzedniej strony.

Dodaliśmy tu jedynie trochę tzw. białych znaków, czyli znaków odstępu i nowego wiersza.

```
setTimeout(function() {
                alert("Już czas wyjąć ciasteczka z piekarnika.");
        }, 600000);
```

Cieszymy się, że poruszyłaś ten problem, gdyż teraz nasz kod jest znacznie bardziej czytelny.

Kup książkę Poleć książkę

Hej, chwilkę... Chyba rozumiem. Ponieważ wyrażenia funkcyjne zwracają referencję do funkcji, zatem możemy używać ich wszędzie tam, gdzie są oczekiwane referencje do funkcji.

Trochę się nagadałeś, ale faktycznie trafiłeś w sedno. To naprawdę jest jedno z kluczowych zagadnień, niezbędnych do zrozumienia, że funkcje są wartościami pierwszej klasy. Jeśli Twój kod oczekuje referencji do funkcji, w tym miejscu zawsze możesz umieścić wyrażenie funkcyjne, ponieważ po przetworzeniu daje ono referencję do funkcji. Jak się właśnie przekonałeś, jeśli argumentem ma być funkcja, nie ma sprawy — możesz zamiast niej przekazać wyrażenie funkcyjne (które przed wykonaniem wywołania zostanie zastąpione referencją do funkcji). To samo dotyczy sytuacji, gdy z jednej funkcji musisz zwrócić inną funkcję — możesz zwrócić wyrażenie funkcyjne.

Kup książkę
Poleć książkę

Ćwiczenie

Teraz upewnimy się, że dobrze zapamiętałeś składnię używaną do przekazywania anonimowych wyrażeń funkcyjnych w wywołaniach innych funkcji. Zmień poniższy kod tak, by zamiast zmiennej (w tym przypadku `vaccine`) argumentem wywołania było anonimowe wyrażenie funkcyjne.

```
function vaccine(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}
administer(patient, vaccine, time);
```

Tu zapisz swoją odpowiedź. I nie zapomnij sprawdzić odpowiedzi, zanim podejmiesz dalszą lekturę!

Nie istnieją głupie pytania

P: Stosowanie funkcji anonimowych w taki sposób wydaje się bardzo zawiłe. Czy naprawdę muszę o tym wiedzieć?

O: Owszem, musisz. Anonimowe wyrażenia funkcyjne bardzo często są używane w kodzie JavaScript, jeśli zatem chcesz nauczyć się analizy kodu napisanego przez innych programistów lub zrozumieć działanie bibliotek JavaScript, musisz wiedzieć, jak one działają i jak je rozpoznawać w kodzie.

P: Czy stosowanie anonimowych wyrażeń funkcyjnych jest lepsze? Uważam, że jedynie komplikuje kod i sprawia, że trudno go czytać i analizować.

O: Poczekaj trochę. Po pewnym czasie, kiedy zobaczysz kod, taki jak ten, znacznie łatwiej będzie Ci go analizować, a naprawdę istnieje bardzo wiele sytuacji, w których taka składnia pozwala zmniejszyć złożoność kodu, sprawia, że jest bardziej przejrzysty, a nasze intencje — łatwiejsze do zauważenia. Z drugiej strony, przesadne wykorzystanie tej techniki na pewno może sprawić, że kod będzie trudniejszy do zrozumienia. Jeśli jednak zaczniesz jej używać, po pewnym czasie stanie się łatwiejsza i bardziej przydatna. Na pewno spotkasz się z wieloma przykładami kodu, który w bardzo dużym stopniu korzysta z funkcji anonimowych, zatem dołączenie tej techniki do swojego przybornika z narzędziami programistycznymi jest dobrym pomysłem.

P: Skoro funkcje pierwszej klasy są tak użyteczne, to dlaczego nie ma ich w innych językach programowania?

O: Ależ są (a ludzie, którzy pracują nad językami, w których ich nie ma, zaczynają rozważać ich dodanie). Przykładowo funkcje pierwszej klasy, takie jak w JavaScripcie, są dostępne w językach Scheme i Scala. Inne języki, takie jak PHP, Java (w najnowszej wersji), C# oraz Objective C, udostępniają większość lub niektóre z ich możliwości. Wraz ze wzrostem liczby osób rozpoznających zalety posiadania funkcji pierwszej klasy w używanym języku programowania coraz więcej języków zaczyna je udostępniać. Jednak każdy język robi to nieco inaczej, zatem badając analogiczne możliwości w innych językach, musisz się przygotować na pewne różnice.

Kup książkę Poleć książkę

Kiedy funkcja zostaje zdefiniowana? To zależy...

Jest jeszcze jedna, interesująca rzecz dotycząca funkcji, o której dotąd nie wspominaliśmy. Czy pamiętasz, że przeglądarka przetwarza kod JavaScript dwukrotnie? W ramach pierwszego przebiegu przetwarzane są wszystkie deklaracje funkcji, a odnalezione funkcje zostają zdefiniowane. Natomiast podczas drugiego przebiegu przeglądarka wykonuje kod JavaScript liniowo, od początku do końca; właśnie podczas tego przebiegu są przetwarzane wyrażenia funkcyjne. A to z kolei określa, gdzie i kiedy będziemy mogli wywoływać funkcje w kodzie.

Aby przekonać się, co to wszystko naprawdę oznacza, przeanalizujemy konkretny przykład. Poniżej przedstawiliśmy kod z poprzedniego rozdziału, w którym wprowadziliśmy nieznaczne zmiany. Spróbujmy go wykonać.

WAŻNE: Przeczytaj to zgodnie z kolejnością cyfr. Zacznij od 1, potem przejdź do 2 itd.

1 Zaczynamy od samego początku kodu i szukamy umieszczonych w nim deklaracji.

4 Ponownie zaczynamy do początku kodu, jednak tym razem zaczynamy go wykonywać.

```
var migrating = true;

if (migrating) {
   quack(4);
   fly(4);
}

var fly = function(num) {
   for (var i = 0; i < num; i++) {
      console.log("Latam!");
   }
};

function quack(num) {
   for (var i = 0; i < num; i++) {
      console.log("Kwak!");
   }
}
```

5 Tworzymy zmienną `migrating` i zapisujemy w niej wartość `true`.

Zwróć uwagę, że przenieśliśmy tę instrukcję warunkową z końca kodu na jego początek.

6 Wyrażenie warunkowe ma wartość `true`, więc wykonujemy blok kodu.

7 Pobieramy referencję do funkcji ze zmiennej `quack` i wywołujemy ją, przekazując do niej wartość 4.

8 Pobieramy referencję do funkcji ze zmiennej `fly`... Chwila, ta zmienna jeszcze nie została zdefiniowana!

2 Znaleźliśmy deklarację funkcji. Tworzymy zatem tę funkcję i zapisujemy ją w zmiennej `quack`.

3 Docieramy do końca kodu. Udało się znaleźć tylko jedną deklarację funkcji.

Kup książkę Poleć książkę

Co się właśnie stało? Dlaczego funkcja fly nie była zdefiniowana?

No dobrze, mogliśmy się przekonać, że funkcja `fly` nie jest zdefiniowana, kiedy spróbowaliśmy ją wykonać, ale dlaczego tak się stało? Przecież funkcja `quack` zadziałała bez problemów. Jak już pewnie odgadłeś, funkcja `fly` — w odróżnieniu od funkcji `quack`, która została zdefiniowana podczas pierwszego przebiegu przetwarzania kodu, gdyż została utworzona przy użyciu deklaracji — jest definiowana podczas drugiego przebiegu, w trakcie którego kod jest wykonywany od początku do końca. Jeszcze raz przyjrzyjmy się naszemu przykładowi.

Konsola JavaScript

```
Kwak!
Kwak!
Kwak!
Kwak!
TypeError: undefined is not a function
```

Kiedy będziemy przetwarzać ten kod i spróbujemy wywołać quack, wszystko zadziała zgodnie z oczekiwaniami, gdyż funkcja quack została zdefiniowana podczas pierwszego przebiegu przetwarzania kodu.

To się dzieje, kiedy spróbujemy wywołać funkcję, która nie jest zdefiniowana.

```
var migrating = true;
if (migrating) {
    quack(4);
    fly(4);
}
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Latam!");
    }
};
function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Kwak!");
    }
}
```

Jednak kiedy spróbujemy wywołać funkcję fly, zostanie wyświetlony błąd, gdyż funkcja ta nie została jeszcze zdefiniowana...

... a zostanie zdefiniowana dopiero w momencie wykonania tej instrukcji, czyli po wywołaniu funkcji fly.

Możesz zobaczyć komunikat o błędzie przypominający przedstawiony tutaj (jego postać zależy od używanej przeglądarki): TypeError: Property 'fly' of object [object Object] is not a function.

Co to wszystko oznacza? Zacznijmy od tego, że oznacza to, iż deklaracje funkcji można umieszczać w dowolnym miejscu kodu — na jego początku, końcu oraz pośrodku — a ich wywołania także mogą być umieszczane w dowolnych miejscach. Deklaracje tworzą funkcje, które są zdefiniowane w całym kodzie (rozwiązanie to jest określane jako *podnoszenie* lub *windowanie*, ang. *hoisting*).

Oczywiście w przypadku wyrażeń funkcyjnych sprawa wygląda inaczej, gdyż one nie będą zdefiniowane, aż do momentu, gdy zostaną wykonane. A zatem jeśli nawet przypiszesz wyrażenie funkcyjne zmiennej globalnej, jak zrobiliśmy w przypadku zmiennej `fly`, nie będziesz mógł użyć jej do wywołania funkcji, aż do momentu, gdy zostanie ona zdefiniowana.

I jeszcze jedno. Obie funkcje w powyższym przykładzie mają *zasięg globalny*, co oznacza, że kiedy już zostaną zdefiniowane, będą widoczne w całym kodzie. Trzeba także pamiętać o funkcjach zagnieżdżonych — czyli funkcjach definiowanych wewnątrz innych funkcji — gdyż ma to wpływ na ich zasięg. Zobacz sam…

Kup książkę Poleć książkę

Zagnieżdżanie funkcji

Definiowanie funkcji wewnątrz innej funkcji jest całkowicie dopuszczalne. Oznacza to, że wewnątrz jednej funkcji można umieścić deklarację innej lub wyrażenie funkcyjne. A jak to działa? Oto krótka odpowiedź na to pytanie: jedyną różnicą pomiędzy funkcją zdefiniowaną na najwyższym poziomie kodu a funkcją zdefiniowaną wewnątrz innej jest ich zasięg. Innymi słowy, umieszczenie jednej funkcji wewnątrz innej ma wpływ na to, w których miejscach kodu będzie ona widoczna.

Aby zrozumieć to zagadnienie, rozszerzymy nieco nasz przykład, dodając do niego zagnieżdżone deklaracje funkcji oraz wyrażenia funkcyjne.

```
var migrating = true;
var fly = function(num) {
    var sound = "Latam!";
    function wingFlapper() {
        console.log(sound);
    }
    for (var i = 0; i < num; i++) {
        wingFlapper();
    }
};
function quack(num) {
    var sound = "Kwak!";
    var quacker = function() {
        console.log(sound);
    };
    for (var i = 0; i < num; i++) {
        quacker();
    }
}
if (migrating) {
    quack(4);
    fly(4);
}
```

Tutaj dodajemy deklarację funkcji o nazwie wingFlapper, umieszczoną wewnątrz wyrażenia funkcyjnego fly.

A tutaj ją wywołujemy.

Tu dodajemy wyrażenie funkcyjne, którego wynik jest zapisywany w zmiennej quacker; przy czym zarówno wyrażenie, jak i zmienna są umieszczone wewnątrz deklaracji funkcji quack.

A tutaj ją wywołujemy.

Przenieśliśmy ten blok kodu na sam koniec, zatem wywołanie funkcji fly nie będzie już przysparzać problemów.

Ćwiczenie

Weź ołówek i zaznacz, jakie fragmenty powyższego kodu obejmuje zasięg funkcji `fly`, `quack`, `wingFlapper` oraz `quacker`. Zaznacz także, które fragmenty kodu obejmuje zasięg tych funkcji, lecz jednocześnie funkcje te nie są tam zdefiniowane.

Kup książkę Poleć książkę

Jaki wpływ na zasięg ma zagnieżdżanie funkcji?

Funkcje zdefiniowane na głównym poziomie kodu mają zasięg globalny, natomiast funkcje zdefiniowane wewnątrz innych funkcji mają zasięg lokalny. Przeanalizujemy kod przedstawiony na poprzedniej stronie i sprawdzimy, jaki zasięg mają poszczególne funkcje. Jednocześnie zastanowimy się, gdzie każda z tych funkcji zostanie zdefiniowana (bądź też, gdzie będzie niezdefiniowana).

```
var migrating = true;
var fly = function(num) {
    var sound = "Latam!";
    function wingFlapper() {
        console.log(sound);
    }
    for (var i = 0; i < num; i++) {
        wingFlapper();
    }
};
function quack(num) {
    var sound = "Kwak!";
    var quacker = function() {
        console.log(sound);
    };
    for (var i = 0; i < num; i++) {
        quacker();
    }
}
if (migrating) {
    quack(4);
    fly(4);
}
```

Wszystko, co zostało zdefiniowane na najwyższym poziomie kodu, ma zasięg globalny. Dlatego zarówno fly, jak i quacker będą zmiennymi globalnymi.

Pamiętaj jednak, że funkcja fly będzie zdefiniowana dopiero po przetworzeniu wyrażenia funkcyjnego.

Funkcja wingFlapper jest definiowana przy użyciu deklaracji umieszczonej wewnątrz funkcji fly. Zatem jej zasięg obejmuje całą funkcję fly i będzie ona zdefiniowana w całym obszarze ciała tej funkcji.

Funkcja quacker jest definiowana przy użyciu wyrażenia funkcyjnego, umieszczonego wewnątrz funkcji quack. A zatem jej zasięg obejmuje całą funkcję quack, jednak będzie ona zdefiniowana od momentu przetworzenia wyrażenia funkcyjnego aż do końca funkcji quack.

Funkcja quacker będzie zdefiniowana tylko w tym fragmencie kodu.

Zauważ, że reguły określające, kiedy można się odwoływać do funkcji, są takie same wewnątrz funkcji, jak i na poziomie globalnym. A zatem jeśli jesteśmy wewnątrz funkcji i funkcja zagnieżdżona została zdefiniowana *przy użyciu deklaracji*, to funkcja ta będzie zdefiniowana w całym obszarze funkcji zewnętrznej. Z drugiej strony, jeśli funkcja wewnętrzna została zdefiniowana *przy użyciu wyrażenia funkcyjnego*, będzie ona zdefiniowana wyłącznie po jego przetworzeniu.

Nie istnieją głupie pytania

P: Kiedy przekazujemy wyrażenie funkcyjne do innej funkcji, to przekazywana funkcja musi zostać zapisana w parametrze, a następnie, wewnątrz funkcji zewnętrznej, jest traktowana jako zmienna lokalna. Czy tak?

O: Dokładnie tak. Przekazywanie funkcji jako argumentu wywołania innej funkcji powoduje skopiowanie referencji do funkcji przekazywanej i zapisanie jej w zmiennej parametru funkcji wywoływanej. A taki parametr zawierający referencję do funkcji, jak każdy inny parametr, jest zmienną lokalną.

Kup książkę Poleć książkę

JAVASCRIPTOWE WYZWANIE EKSTREMALNE

Potrzebujemy eksperta do spraw funkcji pierwszej klasy i słyszeliśmy, że Ty nim jesteś! Poniżej znajdziesz dwa fragmenty kodu i musisz nam pomóc w określeniu, co wykonują, bo utknęliśmy. Dla nas oba fragmenty wyglądają prawie identycznie, z tą różnicą, że jeden używa funkcji pierwszej klasy, a drugi nie. Na podstawie naszej wiedzy o zasięgu w języku JavaScript oczekiwaliśmy, że próbka nr 1 zwróci wartość 008, a próbka nr 2 wartość 007. A jednak obie próbki zwracają wartość 008! Czy możesz nam pomóc w zrozumieniu, dlaczego tak się dzieje?

Sugerujemy, żebyś przyjął zdecydowaną opinię, zapisał ją na tej stronie, a następnie odwrócił kartkę.

Próbka nr 1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}

getSecret();
```

Próbka nr 2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Jeszcze nie patrz na rozwiązanie zamieszczone pod koniec rozdziału, wrócimy do tego wyzwania nieco później.

Kup książkę
Poleć książkę

Krótka powtórka z zasięgu leksykalnego

Leksykalny oznacza, że zasięg zmiennej można określić na podstawie analizy struktury kodu, a nie trzeba z tym czekać do momentu jego realizacji.

Skoro już jesteśmy przy temacie zasięgu, powtórzymy jeszcze raz informacje dotyczące działania zasięgu leksykalnego.

Tu mamy zmienną globalną o nazwie justAVar.

```
var justAVar = "Och, nie przejmuj się, jestem zmienną GLOBALNĄ!";

function whereAreYou() {
    var justAVar = "Szara, zwyczajna zmienna LOKALNA.";

    return justAVar;
}

var result = whereAreYou();
console.log(result);
```

A ta funkcja definiuje nowy zasięg leksykalny...

...w którym mamy zmienną lokalną o nazwie justAVar, przesłaniającą zmienną globalną o tej samej nazwie.

Kiedy ta funkcja zostanie wywołana, zwraca wartość zmiennej justAVar. Tylko której? Używamy zasięgu leksykalnego, zatem odnajdujemy odpowiednią zmienną justAVar, patrząc na zasięg najbliższej funkcji. Jeśli zmiennej tam nie znajdziemy, szukamy jej w zasięgu globalnym.

A zatem kiedy wywołamy funkcję whereAreYou, zwróci ona wartość lokalnej, a nie globalnej zmiennej justAVar.

Konsola JavaScript

```
Szara, zwyczajna zmienna LOKALNA.
```

A teraz wprowadźmy funkcję zagnieżdżoną.

```
var justAVar = "Och, nie przejmuj się, jestem zmienną GLOBALNĄ!";

function whereAreYou() {
    var justAVar = "Szara, zwyczajna zmienna LOKALNA.";

    function inner() {
        return justAVar;
    }

    return inner();
}

var result = whereAreYou();
console.log(result);
```

Tu jest ta sama funkcja.

Jak wcześniej, przesłaniamy zmienną globalną.

Lecz teraz dysponujemy funkcją zagnieżdżoną, która odwołuje się do zmiennej justAVar. Ale do której? Także w tym przypadku używamy zmiennej z najbliższej funkcji zawierającej dany fragment kodu. A zatem użyjemy tej samej zmiennej, co w poprzednim przykładzie.

Zauważ, że funkcję inner wywołujemy w tym miejscu i zwracamy jej wynik.

A zatem kiedy wywołamy funkcję whereAreYou, zostanie wywołana funkcja inner, która zwróci wartość lokalnej, a nie globalnej zmiennej justAVar.

Konsola JavaScript

```
Szara, zwyczajna zmienna LOKALNA.
```


Kup książkę Poleć książkę

Miejsce, w którym zasięg leksykalny sprawia, że sprawy stają się interesujące

Wprowadźmy jeszcze jedną, małą modyfikację. Uważnie przyjrzyj się temu przykładowi, jest naprawdę trudny.

```
var justAVar = "Och, nie przejmuj się, jestem zmienną GLOBALNĄ!";

function whereAreYou() {
    var justAVar = "Szara, zwyczajna zmienna LOKALNA.";

    function inner() {
        return justAVar;
    }

    return inner;
}

var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

Tu nie wprowadziliśmy żadnych zmian, to te same zmienne i funkcje, co wcześniej.

Jednak zamiast wywoływać funkcję inner, tym razem ją zwracamy.

A zatem kiedy wywołamy whereAreYou, uzyskamy referencję do funkcji inner, którą zapisujemy w zmiennej innerFunction. Następnie wywołujemy tę funkcję, zapisujemy zwrócony przez nią wynik w zmiennej result, po czym go wyświetlamy.

Kiedy zatem funkcja inner zostanie wywołana (jako innerFunction) w tym miejscu, której zmiennej justAVar użyje? Tej lokalnej czy globalnej?

Znaczenie ma moment wywołania funkcji. Wywołujemy inner po jej zwróceniu, kiedy w bieżącym zasięgu dostępna jest globalna zmienna justAVar, a zatem zostanie wyświetlony łańcuch „Och, nie przejmuj się, jestem zmienną GLOBALNĄ!".

Nie tak szybko. W zasięgu leksykalnym znaczenie ma struktura, w której funkcja została zdefiniowana, a zatem wynikiem wywołania musi być wartość zmiennej lokalnej, czyli „Szara, zwyczajna zmienna LOKALNA.".

Kup książkę
Poleć książkę

Franek: Co masz na myśli, mówiąc, że to wy macie rację? To tak, jakbyście definiowali prawa fizyki albo coś takiego. Ta zmienna lokalna już nawet nie istnieje… Chodzi mi o to, że kiedy kończy się zasięg zmiennej, przestaje ona istnieć. Po prostu znika! Nie oglądałeś filmu TRON?

Judyta: Tak może jest w Twoim słabym C++ lub w Javie, ale nie w JavaScripcie.

Kuba: Poważnie, jak to możliwe? Funkcja `whereAreYou` została wywołana i tyle, a zmienna lokalna `justAVar` nie może już przecież istnieć?

Judyta: Gdybyście słuchali, co do was mówię… JavaScript nie działa w taki sposób.

Franek: No dobra, rzuć nam jakąś garść informacji. Jak to działa?

Judyta: Kiedy definiujemy funkcję `inner`, zmienna `justAVar` znajduje się w jej zasięgu. Zasięg leksykalny oznacza, że istotny jest sposób definiowania zmiennych; jeśli zatem używamy zasięgu leksykalnego, *to zawsze wtedy, gdy wywołamy funkcję* `inner`, przyjmie ona, że ta zmienna lokalna wciąż jest dla niej dostępna i może z niej korzystać.

Franek: Ale, jak już powiedziałem, to wygląda tak, jakbyśmy zmieniali definicję praw fizyki. Funkcja `whereAreYou`, która zdefiniowała lokalną wersję zmiennej `justAVar`, została już wykonana i przestała istnieć.

Judyta: To prawda. Funkcja `whereAreYou` została wykonana, lecz funkcja `inner` wciąż może skorzystać z jej zasięgu.

Kuba: Ale jak?

Judyta: No dobrze, zobaczmy zatem, co NAPRAWDĘ się dzieje, kiedy definiujemy i zwracamy funkcję…

UWAGA REDAKCYJNA: Czy Józek naprawdę zmienił koszulkę na tej stronie?

Kup książkę Poleć książkę

Funkcje raz jeszcze

Musimy coś wyznać. Nie powiedzieliśmy Ci wszystkiego o funkcjach. Nawet kiedy zapytałeś o to, na co faktycznie wskazują referencje do funkcji, uprościliśmy nieco odpowiedź. Powiedzieliśmy coś w stylu: „Na coś, co przypomina skrystalizowaną funkcję, zawierającą jej blok kodu”.

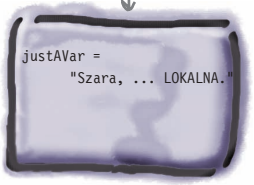
Teraz jednak nadszedł czas, by wszystko wyjaśnić.

W tym celu przeanalizujemy to, co naprawdę dzieje się podczas wykonywania tego kodu; zaczniemy od funkcji `whereAreYou`.

```
function whereAreYou() {
    var justAVar = "Szara, zwyczajna zmienna LOKALNA.";

    function inner() {
        return justAVar;
    }

    return inner;
}
```

1. Najpierw odnajdujemy zmienną lokalną o nazwie `justAVar`. Zapisujemy w niej łańcuch znaków `"Szara, zwyczajna zmienna LOKALNA."`.

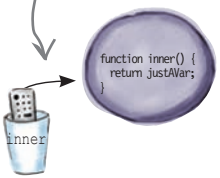
2. Nie wspominaliśmy o tym wcześniej, jednak wszystkie zmienne lokalne są przechowywane w środowisku.



To jest środowisko. Zawiera ono wszystkie zmienne zdefiniowane w zasięgu lokalnym.

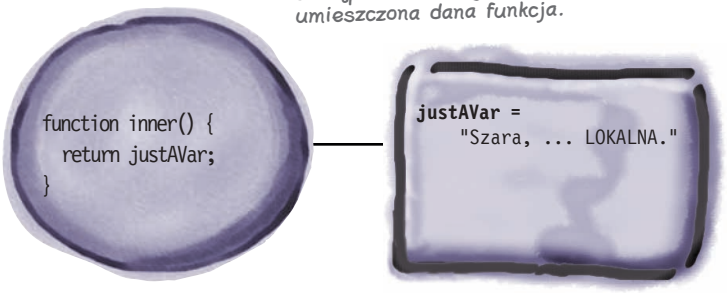
W tym przykładzie środowisko zawiera tylko jedną zmienną, justAVar.

3. Następnie tworzymy funkcję o nazwie `inner`.



4. A później, kiedy zwracamy tę funkcję, okazuje się, że zwracamy nie samą funkcję, lecz funkcję wraz z środowiskiem, które jest z nią skojarzone.

Każda funkcja posiada skojarzone z nią środowisko, które zawiera zmienne lokalne dostępne w zasięgu, w którym jest umieszczona dana funkcja.



Zobaczmy zatem, jak to środowisko jest używane w momencie wywołania funkcji inner...

Kup książkę
Poleć książkę

Wywoływanie funkcji (po raz wtóry)

Skoro już dysponujemy funkcją `inner` oraz jej środowiskiem, spróbujmy ją jeszcze raz wywołać i zobaczyć, co się stanie. Poniżej zamieściliśmy kod, który chcemy wykonać.

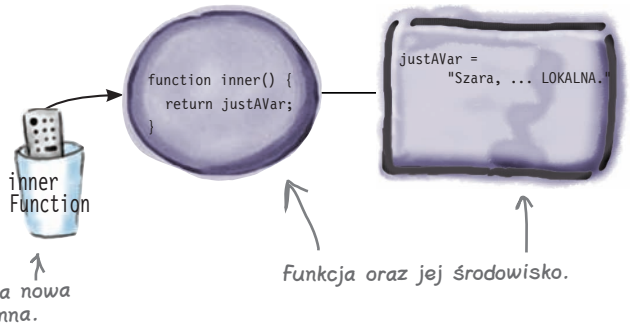
```
var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

---

1 **Najpierw wywołujemy funkcję `whereAreYou`. Już wiemy, że zwraca ona referencję do funkcji. Tworzymy zatem zmienną `innerFunction` i zapisujemy w niej tę zwróconą funkcję. Pamiętaj, że referencja do funkcji jest skojarzona ze środowiskiem.**

```
var innerFunction = whereAreYou();
```

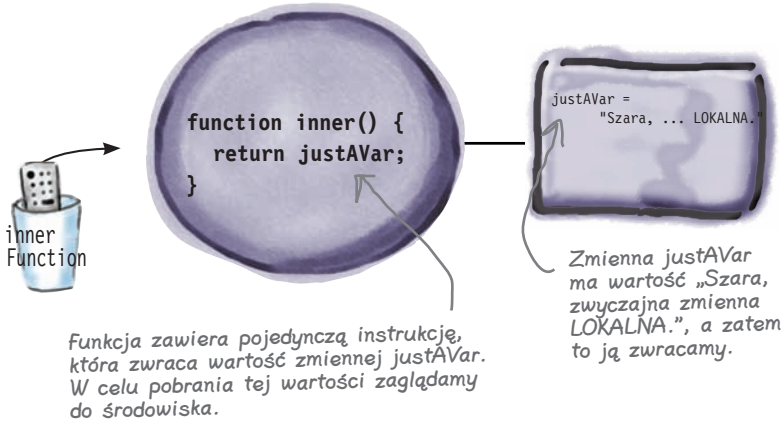
Po wykonaniu tej instrukcji będziemy dysponować zmienną innerFunction odwołującą się do funkcji (oraz środowiskiem zwróconym z funkcji whereAreYou).



---

2 **Następnie wywołujemy funkcję `innerFunction`. W tym celu przetwarzamy kod umieszczony w ciele tej funkcji, a co więcej, robimy to w kontekście jej środowiska.**

```
var result = innerFunction();
```



Kup książkę Poleć książkę

3 **I w końcu zapisujemy wartość zwróconą przez funkcję w zmiennej result i wyświetlamy ją w oknie konsoli.**

```
var result = innerFunction();
console.log(result);
```

Funkcja innerFunction zwraca łańcuch „Szara, zwyczajna zmienna LOKALNA.", który pobraliśmy ze środowiska, zatem zapisujemy go w zmiennej result.

Teraz musimy tylko wyświetlić ten łańcuch w oknie konsoli.

Konsola JavaScript

```
Szara, zwyczajna zmienna LOKALNA
```

Kup książkę
Poleć książkę

Nie istnieją głupie pytania

P: Co mieliście na myśli, pisząc, że zasięg leksykalny określa, gdzie zmienne będą zdefiniowane?

O: Pisząc o zasięgu leksykalnym, mieliśmy na myśli to, że w języku JavaScript reguły określania zasięgu bazują wyłącznie na strukturze kodu (a nie na dynamicznych właściwościach określanych w trakcie wykonywania skryptu). Oznacza to, że obszar, w którym zmienna będzie zdefiniowana, można określić poprzez analizę struktury kodu. Pamiętaj także, że w języku JavaScript jedynie funkcje wprowadzają nowy zasięg. A zatem, widząc odwołanie do zmiennej, należy poszukać, w której funkcji ta zmienna została zdefiniowana, a zaczynać trzeba od funkcji najbardziej zagnieżdżonej i podążać do najbardziej zewnętrznej. Jeśli zmiennej nie udało się znaleźć w żadnej funkcji, oznacza to, że jest zmienną globalną lub nie została wcześniej zdefiniowana.

P: Jak działa środowisko, w przypadku gdy funkcja została zdefiniowana głęboko wewnątrz wielu innych zagnieżdżonych funkcji?

O: Opisując, czym jest środowisko, podaliśmy dosyć uproszczone informacje, jednak możesz to sobie wyobrażać w taki sposób, że każda zagnieżdżona funkcja ma swoje własne środowisko, z własnymi zmiennymi. Następnie tworzony jest łańcuch środowisk wszystkich zagnieżdżonych funkcji, zaczynając od tej najbardziej wewnętrznej, aż do zupełnie zewnętrznej.

Kiedy zatem przychodzi do odnajdywania zmiennej w środowisku, poszukiwania zaczynają się w środowisku najbliższym, a następnie postępują wzdłuż łańcucha, aż do momentu odnalezienia zmiennej. A jeśli nie uda się jej znaleźć, na końcu sprawdzane jest środowisko globalne.

P: Dlaczego zasięg leksykalny i środowiska funkcji są dobrymi rozwiązaniami? Sądziłem, że w przedstawionym przykładzie prawidłową odpowiedzią będzie `"Och, nie przejmuj się, jestem zmienną GLOBALNĄ!"`. Dla mnie to byłoby bardziej sensowne rozwiązanie. Faktyczny stan rzeczy wydaje się mylący i nieintuicyjny.

O: Rozumiemy, że tak możesz uważać, jednak zaletą zasięgu leksykalnego jest to, że zawsze możemy spojrzeć na kod, by określić zasięg, w jakim zmienna została zdefiniowana, i na jego podstawie ustalić jej wartość. A jak już pokazaliśmy, rozwiązanie to obowiązuje zawsze, nawet wtedy, kiedy funkcja jest zwracana z innej i wywoływana znacznie później, w miejscu znajdującym się poza jej początkowym zasięgiem.

Jednak istnieje także inny powód, dla którego można uznać, że jest to dobre rozwiązanie — są nim bardzo interesujące możliwości, jakie ono zapewnia. Zajmiemy się nimi już niedługo.

P: Czy zmienne parametrów także są uwzględniane w środowisku?

O: Tak. Jak już zaznaczyliśmy, parametry można uznawać za zmienne lokalne istniejące wewnątrz funkcji, dlatego też wchodzą w skład środowiska.

P: Czy muszę dokładnie rozumieć, jak działają środowiska?

O: Nie. Musisz natomiast rozumieć reguły zasięgu leksykalnego związane ze zmiennymi, które opisaliśmy w tym rozdziale. Jednak teraz już wiesz, że jeśli masz funkcję zwróconą z innej funkcji, cały czas dysponuje ona swoim początkowym środowiskiem.

Pamiętaj, że w języku JavaScript funkcje zawsze są przetwarzane w tym samym środowisku określającym zasięg, w którym zostały zdefiniowane. Jeśli wewnątrz funkcji chcemy określić, skąd pochodzi zmienna, trzeba przejrzeć funkcje, w których została zagnieżdżona, od najbardziej wewnętrznej do najbardziej zewnętrznej.

Kup książkę Poleć książkę

Czym właściwie są domknięcia?

No jasne, wszyscy mówią o domknięciach jak o *absolutnie niezbędnej* możliwości języka, ale ile osób wie, czym one są i jak ich używać? Naprawdę niewiele. To możliwość, którą wszyscy chcą zrozumieć i którą chciałyby dysponować wszystkie tradycyjne języki programowania.

Spróbujmy przedstawić problem. Zgodnie z tym, co twierdzi wiele dobrze wykształconych osób z branży, *domknięcia są trudne*. Jednak dla Ciebie nie będą dużym problemem. A wiesz dlaczego? Nie, nie… To nie ma nic wspólnego z tym, że ta książka jest „przyjazna dla mózgu", ani z tym, że dysponujemy odlotową aplikacją, którą trzeba było napisać, żeby nauczyć Cię wszystkiego o domknięciach. Przyczyna leży gdzie indziej, Ty *już je poznałeś*. Tylko nie pisaliśmy, że to są domknięcia.

A zatem, bez zbędnego zamieszania podajemy superformalną definicję domknięć.

Jeśli dobrze Cię wyszkoliliśmy, powinieneś sobie w tym momencie myśleć: „O tak, to jest wiedza warta «dużej podwyżki»".

> **Domknięcie, rzeczownik.** Domknięcie jest funkcją skojarzoną z odwołującym się do niej środowiskiem.

No dobrze, możemy się zgodzić z opinią, że ta definicja nie rozwiewa wszystkich wątpliwości. Ale skąd się w ogóle wzięła taka nazwa jak *domknięcie* (ang. *closure*)? Przeanalizujemy to szybko, gdyż faktycznie może to być jedno z tych kluczowych pytań, które może zaważyć na losach rozmowy kwalifikacyjnej lub później na losach podwyżki.

Aby zrozumieć słowo *domknięcie*, należy najpierw zrozumieć ideę *domykania* funkcji.

Zaostrz ołówek

Oto Twoje zadanie: (1) odszukaj i zakreśl wszystkie **zmienne niezależne** występujące w poniższym fragmencie kodu. Zmienna niezależna to termin określający zmienną, która nie została zdefiniowana w zasięgu lokalnym. (2) Wybierz jedno ze środowisk przedstawionych z prawej strony ramki, które **domyka funkcję**. Oznacza to, że środowisko zawiera wartości wszystkich wolnych zmiennych.

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
        ending = " -- Tak tylko mówię!";
    } else if (notSoMuch) {
        ending = " -- Nie za bardzo.";
    }
    alert(phrase + ending);
}
```

Zakreśl zmienne niezależne występujące w tym kodzie — czyli zmienne, które nie zostały zdefiniowane w zasięgu lokalnym.

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paweł";
```

```
beingFunny = true;
justSayin = false;
oocoder = true;
```

```
notSoMuch = true;
phrase = "Hej, la la la";
band = "Policja";
```

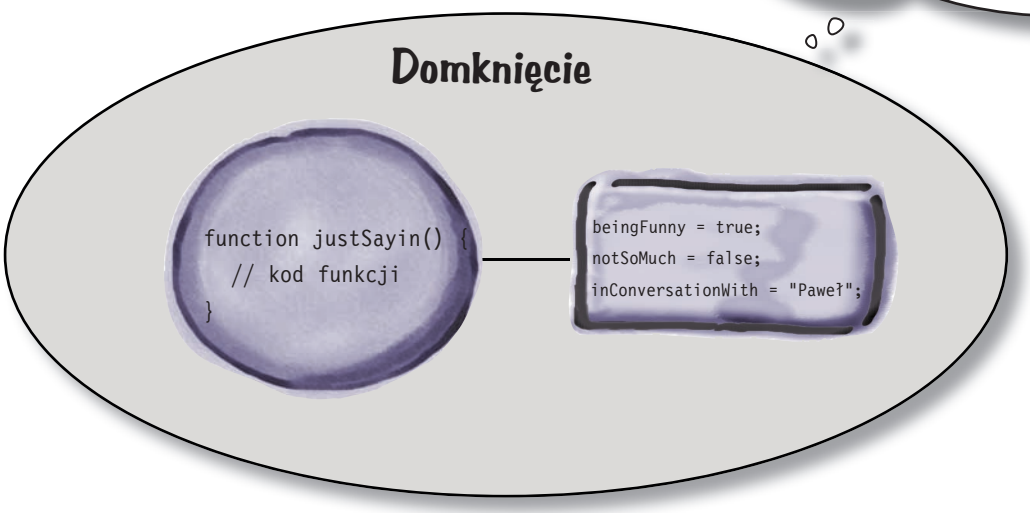
Spośród tych środowisk wybierz to, które domyka funkcję.

Kup książkę Poleć książkę

Domykanie funkcji

Najprawdopodobniej odgadłeś to już, wykonując ostatnie ćwiczenie, jednak przeanalizujemy całe zagadnienie raz jeszcze: funkcja zazwyczaj ma *zmienne lokalne* definiowane w jej ciele (dotyczy to także ewentualnych parametrów), lecz może także używać zmiennych, które nie zostały zdefiniowane lokalnie wewnątrz niej — tzw. *zmiennych niezależnych* (ang. *free variable*). Określenie *niezależne* pochodzi stąd, że wewnątrz ciała funkcji zmiennym niezależnym nie jest przypisana żadna wartość (innymi słowy, nie zostały one zadeklarowane wewnątrz funkcji). A kiedy środowisko zawiera wartości wszystkich zmiennych niezależnych, mówimy, że *domyka* ono funkcję. Idąc dalej, jeśli połączymy funkcję i jej środowisko, uzyskamy *domknięcie* (ang. *closure*).

Jeśli zmienna używana w ciele mojej funkcji nie została zdefiniowana lokalnie i nie jest zmienną globalną, możesz się założyć, że pochodzi ona z jednej z funkcji, w których mnie zagnieżdżono i jest dostępna w moim środowisku.



Domknięcie powstaje, kiedy połączymy funkcję wykorzystującą zmienne niezależne ze środowiskiem udostępniającym wartości, które są powiązane z tymi zmiennymi.

Kup książkę Poleć książkę

To już chyba dziesiąta strona, na której zajmujemy się tym zagadnieniem. Czy kiedyś jeszcze wrócimy do JavaScriptu stosowanego w praktyce? Czy już na zawsze pozostaniemy w świecie teorii? Dlaczego w ogóle mam się przejmować tymi niskopoziomowymi tajnikami działania funkcji? Przecież wystarczy, że będę pisać funkcje i je wywoływać, prawda?

Gdyby tylko domknięcia nie były tak diabelnie przydatne, moglibyśmy się nawet z Tobą zgodzić.

Bardzo nam przykro, że musieliśmy Cię narazić na te wszystkie trudności związane z nauką domknięć, ale zapewniamy, że to się opłaci. Bo widzisz, domknięcia nie są jedynie jakąś teoretyczną konstrukcją programowania funkcyjnego — są bardzo użyteczną techniką programistyczną. A teraz, kiedy już zrozumiałeś, jak działają (i wcale nie żartujemy, pisząc, że dobra znajomość domknięć jest czynnikiem, który może znacząco podnieść Twoje notowania u szefostwa i współpracowników), nadszedł czas, by nauczyć się, jak ich używać.

I jeszcze jedna, kluczowa sprawa: domknięcia są używane wszędzie. W tak dużym stopniu staną się one Twoją drugą naturą, że wkrótce zdasz sobie sprawę, że używasz ich naprawdę bardzo często. Napiszmy w końcu jakieś domknięcie i przekonajmy się, o czym tu w ogóle mówimy.

Kup książkę
Poleć książkę

Zastosowanie domknięć w celu zaimplementowania magicznego licznika

Czy myślałeś kiedyś o zaimplementowaniu funkcji działającej jako licznik? Mogłaby ona wyglądać jakoś tak.

```
var count = 0;

function counter() {
    counter = counter + 1;
    return count;
}
```

Tu mamy zmienną globalną o nazwie count.

Każde wywołanie funkcji counter inkrementuje globalną zmienną count i zwraca jej nową wartość.

Konsola JavaScript

```
1
2
3
```

Taki licznik moglibyśmy wykorzystać w następujący sposób.

```
console.log(counter());
console.log(counter());
console.log(counter());
```

A zatem możemy inkrementować nasz licznik i wyświetlać jego wartość w taki sposób.

Jedynym problemem, jaki możemy wskazać w tym kodzie, jest to, że użyto w nim zmiennej globalnej `count`, co może być problematyczne, gdy nad kodem pracuje zespół programistów (a to dlatego, że ludzie często używają tych samych nazw, co potem powoduje konflikty).

A co byś powiedział na informację, że istnieje sposób zaimplementowania takiego licznika z wykorzystaniem całkowicie lokalnej i chronionej zmiennej `count`? Wtedy zyskałbyś licznik, który nigdy nie będzie kolidował z żadnym innym kodem, a jedynym sposobem inkrementacji jego wartości będzie wywołanie funkcji (inaczej nazywanej domknięciem).

W celu zaimplementowania licznika z użyciem domknięcia możemy wykorzystać przeważającą większość kodu przedstawionego powyżej. Patrz i podziwiaj.

```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
```

Tutaj tworzymy zmienną count wewnątrz funkcji makeCounter. A zatem count jest zmienną lokalną, a nie globalną.

Teraz tworzymy funkcję counter, która inkrementuje wartość zmiennej count.

A tu zwracamy funkcję counter.

To jest domknięcie. A w środowisku funkcji counter przechowywana jest wartość zmiennej count.

Czy sądzisz, że ta magiczna sztuczka się uda? Przekonajmy się…

Kup książkę Poleć książkę

Próbne odliczanie magicznego licznika

Dodaliśmy trochę kodu, który pozwoli przetestować nasz magiczny licznik. Dalej — odpalajmy!

```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
var doCount = makeCounter();
console.log(doCount());
console.log(doCount());
console.log(doCount());
```

Konsola JavaScript

```
1
2
3
```

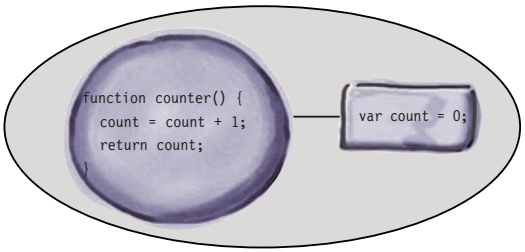
Nasz licznik działa. Uzyskaliśmy prawidłowe wyniki.

Zaglądamy za kulisy...

Przeanalizujemy ten kod krok po kroku, by przekonać się, jak działa.

1. Wywołujemy funkcję `makeCounter`, która tworzy funkcję `counter` i zwraca ją wraz z jej środowiskiem zawierającym zmienną niezależną `count`. Innymi słowy, funkcja ta tworzy domknięcie. Funkcja zwrócona przez funkcję `makeCounter` jest zapisywana w zmiennej `doCount`.
2. Wywołujemy funkcję `doCount`. Wykonuje ona kod umieszczony w ciele funkcji `counter`.
3. Kiedy napotykamy zmienną `count`, szukamy jej w środowisku i pobieramy jej wartość. Wartość tę inkrementujemy, a następnie ponownie zapisujemy w środowisku i zwracamy jako wyniki wykonania funkcji.
4. Powtarzamy kroki 2. i 3., wywołując funkcję `doCount`.

Kiedy wywołujemy funkcję doCount (będącą referencją do funkcji counter), a w niej musimy pobrać wartość zmiennej count, użyjemy zmiennej zapisanej w środowisku domknięcia. Świat zewnętrzny (czyli kod umieszczony w zasięgu globalnym) nigdy nie zobaczy zmiennej count. Jednak my możemy jej używać za każdym razem, kiedy wywołamy funkcję doCount. Co więcej, wywołanie funkcji doCount jest jedynym sposobem odwołania się do zmiennej count.




```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;   ❸
        return count;
    }
    return counter;
}
❶ var doCount = makeCounter();
❷ console.log(doCount());
  console.log(doCount());
❹ console.log(doCount());
```

To jest domknięcie.

Kiedy wywołamy funkcję makeCounter, uzyskamy domknięcie: funkcję skojarzoną ze środowiskiem.

Kup książkę Poleć książkę

Ćwiczenie

Teraz Twoja kolej. Spróbuj utworzyć opisane poniżej domknięcia. Zdajemy sobie sprawę, że początkowo tworzenie domknięć nie jest prostym zadaniem, jeśli zatem będziesz potrzebował pomocy, zajrzyj do odpowiedzi. Ważne jest jednak, żebyś przerobił te przykłady i doszedł do etapu, kiedy całkowicie je zrozumiesz.

Domknięcie pierwsze za 10 punktów. Funkcja `makePassword` pobiera argument reprezentujący hasło i zwraca funkcję, do której przekazywany jest łańcuch porównywany z hasłem. Funkcja ta zwraca wartość `true`, jeśli przekazany łańcuch odpowiada hasłu. (Zanim zrozumiemy działanie domknięcia, czasami trzeba kilkakrotnie przeczytać jego opis).

```
function makePassword(password) {
    return ________________________ {
        return (passwordGuess === password);
    };
}
```

Domknięcie drugie za 20 punktów. Funkcja `multN` pobiera liczbę (nazwijmy ją `n`) i zwraca funkcję. Ta zwrócona funkcja także pobiera liczbę, którą następnie mnoży przez `n` i zwraca jako wynik wywołania.

```
function multN(n) {
    return ________________________ {
        return ____________;
    };
}
```

Domknięcie trzecie za 30 punktów. To jest modyfikacja licznika, który utworzyliśmy na poprzedniej stronie. Funkcja `makeCounter` nie pobiera żadnych argumentów, lecz definiuje zmienną `count`. Następnie tworzy i zwraca obiekt udostępniający jedną metodę, `increment`. Ta metoda inkrementuje wartość zmiennej `count`, a następnie ją zwraca.

Kup książkę Poleć książkę

Tworzenie domknięcia poprzez przekazanie wyrażenia funkcyjnego jako argumentu

Zwracanie funkcji jako wyniku wykonania innej funkcji nie jest jedynym sposobem tworzenia domknięć. Domknięcie budowane jest *zawsze* wtedy, gdy dysponujemy referencją do funkcji, która korzysta ze zmiennych niezależnych i jest wykonywana poza zasięgiem, w jakim została utworzona.

Kolejnym sposobem tworzenia domknięć jest przekazywanie funkcji w wywołaniu innej. Przekazywana funkcja zostanie wykonana w zupełnie innym kontekście niż ten, w którym została zdefiniowana. Poniżej przedstawiliśmy przykład takiego rozwiązania.

```
function makeTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

}

makeTimer("Ciasteczka są już gotowe!", 1000);
```

Tu mamy funkcję...

... wykorzystującą zmienną niezależną...

... której użyjemy do obsługi licznika czasu, uruchamianego za pomocą metody setTimeout

... a nasza funkcja zostanie wykonana po upływie 1000 milisekund od chwili obecnej, czyli na długo po wykonaniu funkcji makeTimer.

W powyższym przykładzie przekazujemy do metody `setTimeout` wyrażenie funkcyjne, korzystające ze zmiennej niezależnej `doneMessage`. Jak już wiesz, przekazywane wyrażenie funkcyjne zostanie przetworzone i zwróci referencję do funkcji, która zostanie przekazana do metody `setTimeout`. Metoda `setTimeout` zapamięta tę funkcję (a właściwie funkcję i jej środowisko, czyli domknięcie), a następnie po 1000 milisekund wywoła ją.

Także w tym przypadku funkcja przekazana w wywołaniu `setTimeout` jest domknięciem, gdyż towarzyszy jej środowisko kojarzące zmienną niezależną, `doneMessage`, z jej wartością — `"Ciasteczka są już gotowe!"`.

WYSIL SZARE KOMÓRKI

A co by się stało, gdyby Twój kod wyglądał tak.

```
function handler() {
    alert(doneMessage);
}
function makeTimer(doneMessage, n) {
    setTimeout(handler, n);
}
makeTimer("Ciasteczka są już gotowe!", 1000);
```

WYSIL SZARE KOMÓRKI²

Przejrzyj jeszcze raz kod przedstawiony na stronie 438 w rozdziale 9.
Czy potrafiłbyś zmodyfikować ten kod, tak by korzystał z domknięcia, i wyeliminować konieczność podawania w wywołaniu metody `setTimeout` trzeciego argumentu?

Kup książkę Poleć książkę

Domknięcia zawierają rzeczywiste środowisko, a nie jego kopię

Jedną z rzeczy, która często przysparza problemów osobom poznającym domknięcia, jest mylna opinia, że środowisko, jakim dysponuje domknięcie, musi zawierać kopie wszystkich zmiennych oraz ich wartości. A tak nie jest. W rzeczywistości środowisko odwołuje się do faktycznych zmiennych używanych w kodzie, jeśli zatem wartość zostanie zmieniona przez jakiś kod umieszczony poza funkcją domknięcia, podczas wykonywania tej funkcji wykorzysta ona tę nową wartość.

Zmodyfikujmy nasz przykład, by przekonać się, co to oznacza.

```
function setTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

    doneMessage = "AUĆ!";
}
setTimer("Ciasteczka są już gotowe!", 1000);
```

Tutaj jest tworzone domknięcie.

A teraz zmieniamy wartość zmiennej doneMessage.

1. Kiedy wywołujemy `setTimeout`, przekazując do niej wyrażenie funkcyjne, tworzone jest domknięcie zawierające funkcję oraz referencję do jej środowiska.

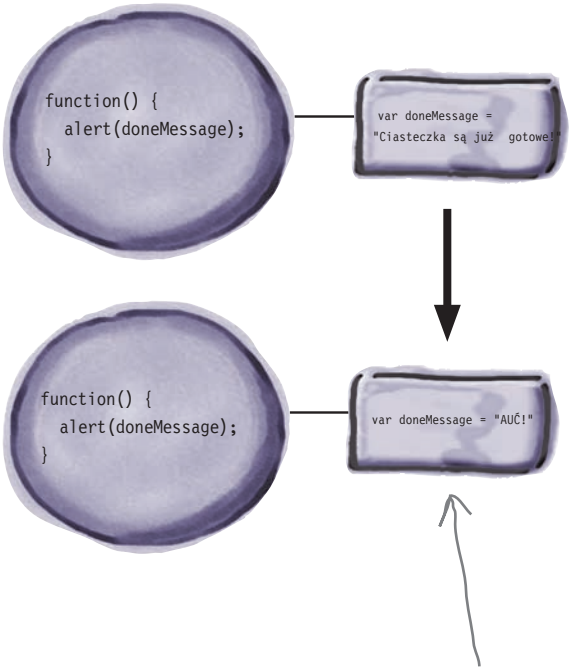
```
setTimeout(function() {
    alert(doneMessage);
}, n);
```

2. Następnie, kiedy poza domknięciem zmienimy wartość parametru `doneMessage` na `"AUĆ!"`, jest ona zmieniana w tym samym środowisku, które jest używane przez domknięcie.

```
doneMessage = "AUĆ!";
```

3. 1000 milisekund później zostaje wywołana funkcja umieszczona w domknięciu. Odwołuje się ona do zmiennej `doneMessage`, której aktualna wartość w środowisku wynosi `"AUĆ!"`. A zatem w oknie dialogowym zostanie wyświetlony łańcuch znaków `"AUĆ!"`.

```
function() { alert(doneMessage); }
```



Kiedy funkcja zostanie wywołana, użyje wartości zmiennej doneMessage zapisanej w środowisku, czyli nowej wartości, którą przypisaliśmy tej zmiennej w funkcji setTimer.

Kup książkę Poleć książkę

Tworzenie domknięć jako procedur obsługi zdarzeń

Przyjrzyjmy się jeszcze jednemu sposobowi tworzenia domknięć. Tym razem zbudujemy domknięcie działające jako procedura obsługi zdarzeń; jest to rozwiązanie dosyć często spotykane w kodzie JavaScript. Zaczniemy od utworzenia prostej strony WWW, zawierającej przycisk oraz element `<div>`, w którym będzie wyświetlany komunikat. Umieszczony na tej stronie skrypt będzie zliczał liczbę kliknięć przycisku i wyświetlał ją w elemencie `<div>`.

Poniżej przedstawiliśmy kody HTML i CSS naszej strony, dodaj je do pliku o nazwie *divClosure.html*.

```
<!doctype html>
<html lang="pl">
<head>
<meta charset="utf-8">
<title>Kliknij mnie!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    // Tu umieścimy kod JavaScript.
</script>
</head>
<body>
    <button id="clickme">Kliknij mnie!</button>
    <div id="message"></div>
</body>
</html>
```

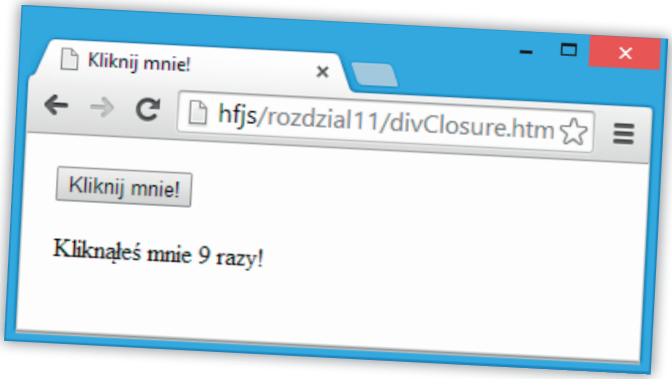
To zwyczajna, typowa strona WWW.

Z prostym arkuszem stylów CSS, służącym do określenia wyglądu elementów.

Tu umieścisz swój kod JavaScript.

Na stronie znajdują się przycisk oraz element prezentujący komunikaty, które będziemy aktualizować po każdym kliknięciu przycisku.

A to jest nasz cel: po każdym kliknięciu przycisku chcemy aktualizować komunikat prezentowany w elemencie i wyświetlać w nim, ile razy przycisk został kliknięty.



A teraz napiszemy kod. Oczywiście mógłbyś napisać kod realizujący to, o co nam chodzi w tym przykładzie, bez stosowania domknięcia, jednak — jak się przekonasz — przy użyciu domknięcia kod będzie bardziej zwarty, a nawet wydajniejszy.

Kup książkę
Poleć książkę

Kliknij mnie! Bez użycia domknięcia

Najpierw pokażemy, jak można zaimplementować kod tego przykładu *bez użycia domknięcia*.

```
var count = 0;

window.onload = function() {
    var button = document.getElementById("clickme");
    button.onclick = handleClick;
};

function handleClick() {
    var message = "Kliknąłeś mnie ";
    var div = document.getElementById("message");
    count++;
    div.innerHTML = message + count + " razy!";
}
```

Zmienna count będzie musiała być zmienną globalną, bo gdyby została zdefiniowana lokalnie wewnątrz funkcji handleClick (czyli procedury obsługi zdarzeń click w przycisku), byłaby inicjalizowana podczas obsługi każdego kliknięcia przycisku.

W funkcji obsługującej zdarzenie load pobieramy element przycisku i określamy procedurę obsługi zdarzeń clik, używając właściwości onclick.

To jest procedura obsługi zdarzeń click naszego przycisku.

Definiujemy zmienną message...

... pobieramy z DOM element ...

... inkrementujemy licznik kliknięć...

... i aktualizujemy zawartość elementu , wyświetlając w nim komunikat o liczbie kliknięć.

Kliknij mnie! Z użyciem domknięcia

Przedstawiona powyżej wersja kodu, w której nie używamy domknięcia, działa doskonale, jej jedynym problematycznym aspektem jest zastosowanie zmiennej globalnej, która potencjalnie może przysporzyć problemów. Zmodyfikujmy zatem kod przykładu, by skorzystać z domknięcia, a następnie spróbujmy porównać oba rozwiązania. Kod pokażemy poniżej, natomiast dokładniejszą analizę jego działania zamieścimy po przetestowaniu.

```
window.onload = function() {
    var count = 0;
    var message = "Kliknąłeś mnie ";
    var div = document.getElementById("message");

    var button = document.getElementById("clickme");
    button.onclick = function() {
        count++;
        div.innerHTML = message + count + " razy!";
    };
};
```

Teraz wszystkie używane zmienne są zmiennymi lokalnymi istniejącymi wewnątrz procedury obsługi window.onload. Nie mogą zatem wystąpić żadne konflikty nazw.

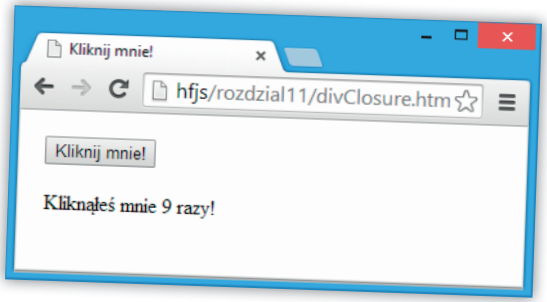
Określamy procedurę obsługi zdarzeń click, którą będzie wyrażenie funkcyjne zapisywane we właściwości onclick przycisku. Dzięki temu wewnątrz tej funkcji możemy odwoływać się do wszystkich zmiennych: div, message oraz count. (Pamiętaj o zasięgu leksykalnym!).

Ta funkcja używa trzech zmiennych niezależnych: div, message oraz count; dlatego też na potrzeby procedury obsługi zdarzeń click tworzone jest domknięcie. A zatem rzeczywiście we właściwości onclick przycisku zapisywane jest właśnie domknięcie.

Kup książkę Poleć książkę

Jazda próbna z licznikiem kliknięć

No dobrze, połączmy zatem kody HTML i JavaScript, zapisując je w jednym pliku, *divClosure.html*, a następnie wypróbujmy działanie strony. Wczytaj stronę w przeglądarce i kliknij przycisk, aby inkrementować wartość licznika. Powinieneś zobaczyć, że komunikat wyświetlony w elemencie `<div>` został zmodyfikowany. Popatrz jeszcze raz na kod i upewnij się, że rozumiesz, jak działa. Kiedy już to zrobisz, przewróć kartkę — na następnej stronie dokładnie analizujemy to rozwiązanie.



Oto rezultat, który uzyskaliśmy.

Uaktualnij kod pliku divClosure.html, by odpowiadał przykładowi przedstawionemu na tej stronie.

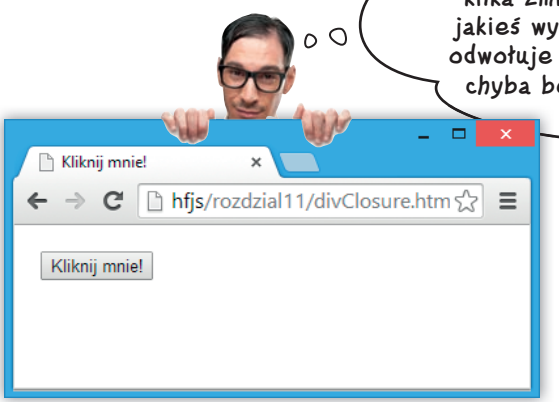
```
<html lang="pl">
<head>
<meta charset="utf-8">
<title>Kliknij mnie!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>

window.onload = function() {
    var count = 0;
    var message = "Kliknąłeś mnie ";
    var div = document.getElementById("message");

    var button = document.getElementById("clickme");
    button.onclick = function() {
        count++;
        div.innerHTML = message + count + " razy!";
    };
};
</script>
</head>
<body>
    <button id="clickme">Kliknij mnie!</button>
    <div id="message"></div>
</body>
</html>
```


Kup książkę
Poleć książkę

Jak działa domknięcie liczące kliknięcia?

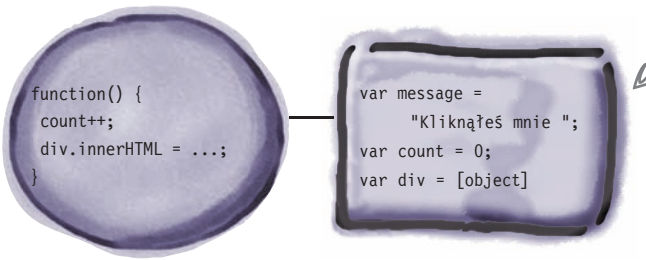
Aby zrozumieć, jak działa to domknięcie, prześledzimy, co robi przeglądarka podczas wykonywania kodu naszego przykładu.

Strona została wczytana, zatem mogę wykonać procedurę obsługi zdarzeń load. Muszę zdefiniować kilka zmiennych... O, widzę, że mam też do przetworzenia jakieś wyrażenie funkcyjne. Zobaczmy... Wygląda na to, że odwołuje się ono do trzech zmiennych niezależnych, zatem chyba będzie lepiej, jeśli utworzę domknięcie.

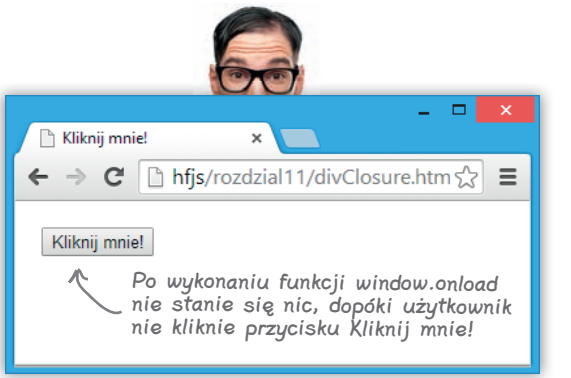



```
window.onload = function() {
    var count = 0;
    var message = "Kliknąłeś mnie ";
    var div = document.getElementById("message");

    var button = document.getElementById("clickme");
    button.onclick = function() {
        count++;
        div.innerHTML = message + count + " razy!";
    };
};
```

Przeglądarka tworzy domknięcie na potrzeby funkcji zapisywanej we właściwości onclick przycisku. Środowisko wchodzące w skład tego domknięcia zawiera trzy zmienne: div, message oraz count.



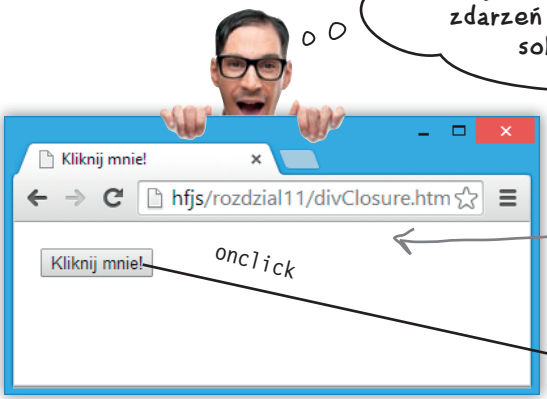
A teraz zapiszę to domknięcie we właściwości onclick przycisku Kliknij mnie!. No dobra, to by było wszystko, co miałam zrobić w ramach obsługi zdarzenia load... Teraz wystarczy, że poczekam, aż ktoś kliknie przycisk na stronie.



Po wykonaniu funkcji window.onload nie stanie się nic, dopóki użytkownik nie kliknie przycisku Kliknij mnie!

Kup książkę Poleć książkę

Hej, ktoś kliknął przycisk! Nadszedł czas, żeby wykonać jego procedurę obsługi zdarzeń click, którą przygotowałam sobie już wcześniej...



Choć zmienna button już nie istnieje (została usunięta po zakończeniu wykonywania funkcji window.onload), jednak obiekt przycisku wciąż egzystuje w DOM, a w jego właściwości onclick jest zapisane nasze domknięcie.

```
function() {
 count++;
 div.innerHTML = ...;
}
```

```
var message =
      "Kliknąłeś mnie ";
var count = 0;
var div = [object]
```

Och, widzę, że mamy tu jakieś domknięcie. No i świetnie, to znaczy, że w jego środowisku znajdę wartości tych trzech zmiennych niezależnych.

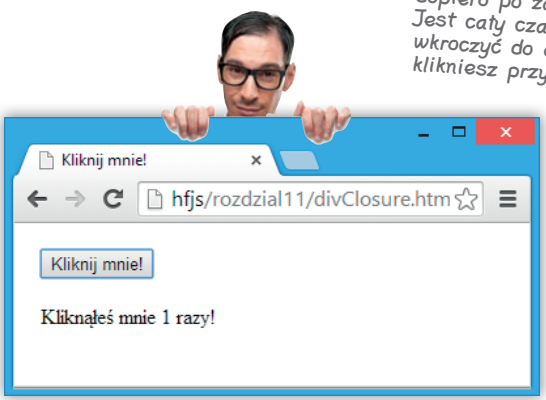
```
function() {
 count++;
 div.innerHTML = ...;
}
```

```
var message =
      "Kliknąłeś mnie ";
var count = 1;
var div = [object]
```

Zauważ, że zmienna div w domknięciu zawiera obiekt. Kiedy inicjalizowaliśmy tę zmienną w funkcji window.onload, zapisaliśmy w niej obiekt zwrócony przez wywołanie metody document.getElementById; dzięki temu teraz nie musimy ponownie pobierać obiektu z DOM, gdyż już nim dysponujemy. To eliminuje jedną operację i sprawia, że nasz kod będzie działał odrobinkę szybciej.

Nasze domknięcie zniknie dopiero po zamknięciu strony. Jest cały czas gotowe, by wkroczyć do akcji, gdy klikniesz przycisk.

Inkrementowałam wartość zmiennej count i zadbałam, by wartość ta została zaktualizowana także w środowisku. Zaktualizowałam także komunikat wyświetlany na stronie... Nie pozostaje mi zatem nic innego, jak tylko czekać na kolejne kliknięcie przycisku.



Kup książkę
Poleć książkę

PO RAZ WTÓRY

JAVASCRIPTOWE WYZWANIE EKSTREMALNE

Potrzebujemy eksperta do spraw domknięć i słyszeliśmy, że Ty nim jesteś. Teraz już wiesz, jak działają domknięcia. Czy zatem potrafisz wyjaśnić, dlaczego obie przedstawione poniżej próbki zwracają wynik 008? Aby to wyjaśnić, zapisz wszelkie zmienne przechowywane w środowiskach funkcji. Zwróć uwagę, że nic nie stoi na przeszkodzie, by środowisko było puste. Sprawdź naszą odpowiedź podaną pod koniec rozdziału.

Próbka nr 1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

Środowisko

Próbka nr 2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Środowisko

Kup książkę Poleć książkę

Zaostrz ołówek

Na początku przyjrzyj się poniższemu fragmentowi kodu.

```
(function(food) {
    if (food === "ciasteczka") {
        alert("Poproszę o więcej.");
    } else if (food === "ciasto") {
        alert("Mniam mniam.");
    }
})("ciasteczka");
```

Wyrażenie funkcyjne umieszczone bezpośrednio w instrukcji wywołania... To już naprawdę ekstremalne rozwiązanie.

Twoim zadaniem jest nie tylko ustalić, jaki będzie wynik wykonania tego fragmentu kodu, lecz przede wszystkim określić, *jak działa*. W tym celu postępuj odwrotnie niż do tej pory, czyli wyodrębnij w kodzie funkcję anonimową, przypisz ją jakieś zmiennej, a następnie umieść tę zmienną tam, gdzie wcześniej było wyrażenie funkcyjne. Czy po takich zmianach kod jest bardziej czytelny? A zatem, co on robi?

Kup książkę
Poleć książkę

CELNE SPOSTRZEŻENIA

- **Funkcja anonimowa** to wyrażenie funkcyjne, które nie ma nazwy.
- Funkcje anonimowe pozwalają na tworzenie bardziej zwartego kodu.
- **Deklaracje funkcji** są definiowane przed wykonaniem pozostałych fragmentów kodu.
- **Wyrażenia funkcyjne** są przetwarzane w trakcie wykonywania pozostałych fragmentów kodu, a zatem nie będą zdefiniowane aż do momentu wykonania instrukcji, w których zostały zapisane.
- Wyrażenie funkcyjne można przekazać do innej funkcji bądź też zwrócić jako wynik wykonania funkcji.
- Wyrażenia funkcyjne zwracają **referencję do funkcji**, co oznacza, że można ich używać wszędzie tam, gdzie można korzystać z referencji do funkcji.
- **Funkcje zagnieżdżone** to funkcje zdefiniowane wewnątrz innych funkcji.
- Funkcje zagnieżdżone mają zasięg lokalny, podobnie jak zmienne lokalne.
- **Zasięg leksykalny** oznacza, że zasięg zmiennej można określić na podstawie analizy kodu.
- W celu powiązania wartości ze zmienną w funkcji zagnieżdżonej używana jest wartość zdefiniowana w najbliższej funkcji zewnętrznej. Jeśli nie uda się znaleźć takiej wartości, sprawdzany jest zasięg globalny.
- **Domknięciem** nazywamy funkcję wraz ze skojarzonym z nią środowiskiem.
- Domknięcia zawierają wartości zmiennych dostępnych w danym zasięgu w momencie tworzenia tego domknięcia.
- **Zmiennymi niezależnymi** stosowanymi w ciele funkcji są zmienne, które w danej funkcji nie są powiązane z żadną wartością.
- Jeśli funkcja domknięcia zostanie wywołana w innym kontekście niż ten, w którym została utworzona, wartości zmiennych niezależnych są pobierane ze środowiska wchodzącego w skład domknięcia.
- Domknięcia są często stosowane do zapamiętywania stanu w procedurach obsługi zdarzeń.

Kup książkę Poleć książkę

Zaostrz ołówek
Rozwiązanie

Przedstawiony poniżej fragment kodu zapewnia kilka możliwości zastosowania funkcji anonimowych. Skorzystaj z nich i użyj funkcji anonimowych wszędzie tam, gdzie to możliwe. Możesz przekreślić stary kod i obok napisać nowy. I jeszcze jedna rzecz: zakreśl wszystkie anonimowe funkcje, które już są używane. Poniżej zamieściliśmy rozwiązanie.

```
window.onload = init;
var cookies = {
    instructions: "Wstępne rozgrzewanie do 175 stopni...",
    bake: function(time) {
            console.log("Wypiekam ciasteczka.");
            setTimeout(done, time);
        }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Już można wypiekać ciasteczka.");
    cookies.bake(2500);
}
function done() {
    alert("Ciasteczka są gotowe, wyciągnij je, by przestygły.");
    console.log("Chłodzenie ciasteczek.");
    var cool = function() {
        alert("Ciasteczka są już zimne, można je jeść!");
    };
    setTimeout(cool, 1000);
}
```


Kup książkę
Poleć książkę

Zmodyfikowaliśmy kod, by utworzyć dwa anonimowe wyrażenia funkcyjne — jedno zastępujące funkcję init, a drugie — funkcję handleButton.

Teraz przypisujemy wyrażenie funkcyjne właściwości window.onload...

```
window.onload = function () {
    var button = document.getElementById("bake");
    button.onclick = function() {
        console.log("Już można wypiekać ciasteczka.");
        cookies.bake(2500);
    }
};

var cookies = {
    instructions: "Wstępne rozgrzewanie do 175 stopni...",
    bake: function(time) {
            console.log("Wypiekam ciasteczka.");
            setTimeout(done, time);
        }
};

function done() {
    alert("Ciasteczka są gotowe, wyciągnij je, by przestygły.");
    console.log("Chłodzenie ciasteczek.");
    var cool = function() {
        alert("Ciasteczka są już zimne, można je jeść!");
    };
    setTimeout(cool, 1000);
}
```

...a drugie wyrażenie funkcyjne przypisujemy właściwości button.onclick.

Należą Ci się dodatkowe słowa uznania, jeśli zauważyłeś, że funkcję cool można umieścić bezpośrednio w wywołaniu metody setTimeout, tak jak pokazaliśmy poniżej.

```
setTimeout(function() {
    alert("Ciasteczka są już zimne, można je jeść!");
}, 1000);
```


Kup książkę Poleć książkę

Ćwiczenie Rozwiązanie

Teraz upewnimy się, że dobrze zapamiętałeś składnię używaną do przekazywania anonimowych wyrażeń funkcyjnych w wywołaniach innych funkcji. Zmień poniższy kod tak, by zamiast zmiennej (w tym przypadku `vaccine`) argumentem wywołania było anonimowe wyrażenie funkcyjne. Oto nasze rozwiązanie.

```
administer(patient, function(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}, time);
```

Zauważ, że nic nie stoi na przeszkodzie, by wyrażenie funkcyjne przekazywane jako argument zapisywać w kilku wierszach. Jednak uważaj na składnię, w takich sytuacjach bardzo łatwo można popełnić błąd!

Ćwiczenie Rozwiązanie

Teraz Twoja kolej. Spróbuj utworzyć opisane poniżej domknięcia. Zdajemy sobie sprawę, że tworzenie domknięć nie jest prostym zadaniem.

Poniżej przedstawiliśmy nasze rozwiązanie:

Domknięcie pierwsze za 10 punktów. Funkcja `makePassword` pobiera argument reprezentujący hasło i zwraca funkcję, do której przekazywany jest łańcuch porównywany z hasłem. Funkcja ta zwraca wartość `true`, jeśli przekazany łańcuch odpowiada hasłu. (Zanim zrozumiemy działanie domknięcia, czasami trzeba kilkakrotnie przeczytać jego opis).

```
function makePassword(password) {
    return function guess(passwordGuess) {
        return (passwordGuess === password);
    };
}

var tryGuess = makePassword("tajne");
console.log("Próbuję 'nic z tego': " + tryGuess("nic z tego"));
console.log("Próbuję 'tajne': " + tryGuess("tajne"));
```

Wynikiem wywołania funkcji makePassword jest domknięcie, w którego skład wchodzi środowisko zawierające zmienną niezależną password.

W wywołaniu funkcji makePassword przekazujemy słowo „tajne", a zatem to właśnie ten łańcuch znaków zostanie zapisany w środowisku domknięcia.

Zauważ, że używamy tu wyrażenia z funkcją, która ma nazwę! Nie jest to konieczne, lecz jest wygodne, gdyż pozwala odwoływać się do funkcji wewnętrznej przy użyciu nazwy. Dodatkowo zwróć uwagę, że zwróconą funkcję musimy wywoływać za pomocą nazwy tryGuess (a nie guess).

A kiedy wywołujemy funkcję tryGuess, porównujemy przekazany ciąg („nic z tego" lub „tajne") z wartością zmiennej password przechowywaną w środowisku funkcji tryGuess.

Dalsza część rozwiązania znajduje się na następnej stronie...

Kup książkę Poleć książkę

Ćwiczenie Rozwiązanie

Teraz Twoja kolej. Spróbuj utworzyć opisane poniżej domknięcia. Zdajemy sobie sprawę, że początkowo tworzenie domknięć nie jest prostym zadaniem.

Poniżej przedstawiliśmy ciąg dalszy naszego rozwiązania.

Domknięcie drugie za 20 punktów. Funkcja `multN` pobiera liczbę (nazwijmy ją `n`) i zwraca funkcję. Ta zwrócona funkcja także pobiera liczbę, którą następnie mnoży przez `n` i zwraca jako wynik wywołania.

```
function multN(n) {
    return function multBy(m) {
        return n*m;
    };
}
var multBy3 = multN(3);
console.log("Mnożymy liczbę 2: " + multBy3(2));
console.log("Mnożymy liczbę 3: " + multBy3(3));
```

Wynikiem zwracanym przez funkcję multN jest domknięcie, którego środowisko zawiera zmienną niezależną n.

A zatem wywołujemy multN(3) i otrzymujemy funkcję, która mnoży dowolną przekazaną liczbę razy 3.

Domknięcie trzecie za 30 punktów. To jest modyfikacja licznika, który utworzyliśmy wcześniej w tym rozdziale. Funkcja `makeCounter` nie pobiera żadnych argumentów, lecz definiuje zmienną `count`. Następnie tworzy i zwraca obiekt udostępniający jedną metodę, `increment`. Metoda ta inkrementuje wartość zmiennej `count`, a następnie ją zwraca.

```
function makeCounter() {
    var count = 0;
    return {
        increment: function() {
            count++;
            return count;
        }
    };
}
var counter = makeCounter();
console.log(counter.increment());
console.log(counter.increment());
console.log(counter.increment());
```

Ta funkcja przypomina wcześniejszą wersję funkcji makeCounter, z tą różnicą, że teraz zwraca obiekt udostępniający metodę increment, a nie samą funkcję.

Metoda increment używa zmiennej niezależnej, count. A zatem metoda ta jest domknięciem, którego środowisko zawiera wartość zmiennej count.

Teraz wywołujemy funkcję makeCounter i otrzymujemy z powrotem obiekt udostępniający metodę (czyli domknięcie).

Wywołujemy metodę w standardowy sposób, a kiedy to robimy, metoda odwołuje się do zmiennej count przechowywanej w jej środowisku.

Kup książkę Poleć książkę

Zaostrz ołówek
Rozwiązanie

Skorzystaj ze swojej wiedzy na temat funkcji i zmiennych, aby wskazać, które z poniższych stwierdzeń są prawdziwe. Poniżej znajdziesz nasze rozwiązanie:

- ☑ Zmienna handler zawiera referencję do funkcji.
- ☑ Kiedy przypisujemy `handler` właściwości `window.onload`, zapisujemy w niej referencję do funkcji.
- ☑ Jedynym powodem istnienia zmiennej `handler` jest zapisanie jej we właściwości `window.onload`.
- ☑ Już nigdy więcej nie użyjemy funkcji `handler`, gdyż jest to kod, który z założenia ma być wykonywany wyłącznie podczas pierwszego wczytania strony.
- ☑ Dwukrotne wywoływanie procedury obsługi zdarzeń `load` nie jest dobrym pomysłem — może ono doprowadzić do wystąpienia problemów, gdyż ten kod służy zazwyczaj do wykonywania czynności związanych z inicjalizacją całej strony.
- ☑ Wyrażenia funkcyjne tworzą referencje do funkcji.
- ☑ Czy wspominaliśmy, że przypisując funkcję `handler` właściwości `window.onload`, zapisujemy w niej referencję do funkcji?

Zaostrz ołówek
Rozwiązanie

Oto Twoje zadanie: (1) odszukaj i zakreśl wszystkie **zmienne niezależne** występujące w poniższym fragmencie kodu. Zmienna niezależna to termin określający zmienną, która nie została zdefiniowana w zasięgu lokalnym. (2) Wybierz jedno z środowisk przedstawionych z prawej strony ramki, które **domyka funkcję**. Oznacza to, że środowisko zawiera wartości wszystkich wolnych zmiennych. Poniżej znajdziesz nasze rozwiązanie.

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
        ending = " -- Tak tylko mówię!";
    } else if (notSoMuch) {
        ending = " -- Nie za bardzo.";
    }
    alert(phrase + ending);
}
```

Zakreśl zmienne niezależne występujące w tym kodzie — czyli zmienne, które nie zostały zdefiniowane w zasięgu lokalnym.

To środowisko zawiera dwie zmienne niezależne: beingFunny oraz notSoMuch.

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paweł";
```

```
beingFunny = true;
justSayin = false;
oocoder = true;
```

```
notSoMuch = true;
phrase = "Hej, la la la";
band = "Policja";
```

Spośród tych środowisk wybierz to, które domyka funkcję.

Kup książkę Poleć książkę

ROZWIĄZANIE

JAVASCRIPTOWE WYZWANIE EKSTREMALNE

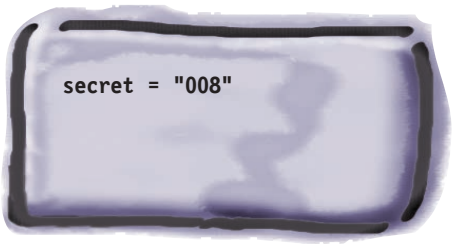
Potrzebujemy eksperta do spraw domknięć i słyszeliśmy, że Ty nim jesteś. Teraz już wiesz, jak działają domknięcia, czy zatem potrafisz wyjaśnić, dlaczego obie przedstawione poniżej próbki zwracają wynik 008? Aby to wyjaśnić, zapisz wszelkie zmienne przechowywane w środowiskach funkcji. Zwróć uwagę, że nic nie stoi na przeszkodzie, by środowisko było puste. Oto nasze rozwiązanie.

Próbka nr 1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

W funkcji getValue secret jest zmienną niezależną...

Środowisko



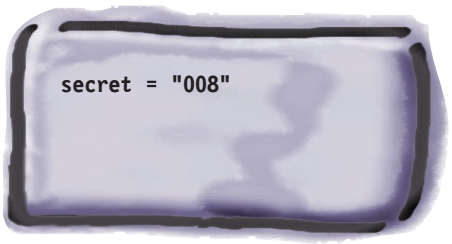
...a zatem zostaje zapisana w środowisku tej funkcji. Jednak funkcja getSecret nie zwraca funkcji getValue, dlatego też nigdy nie zobaczymy domknięcia poza kontekstem, w którym jest tworzone.

Próbka nr 2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

W funkcji getValue secret jest zmienną niezależną...

Środowisko



a w tym przypadku tworzymy domknięcie, które zwracamy jako wynik wywołania funkcji getSecret. Kiedy zatem wywołamy funkcję getValueFun (czyli getValue) w innym kontekście (w tym przypadku: w kontekście globalnym), użyta zostanie wartość zmiennej secret przechowywana w środowisku.

Kup książkę Poleć książkę

Zaostrz ołówek
Rozwiązanie

Oto nasze rozwiązanie tego zakręconego zadania.

```
(function(food) {
    if (food === "ciasteczka") {
        alert("Poproszę o więcej.");
    } else if (food === "ciasto") {
        alert("Mniam mniam.");
    }
})("ciasteczka");
```

Twoim zadaniem jest nie tylko określenie, jaki będzie wynik wykonania tego fragmentu kodu, lecz przede wszystkim, *jak* on działa. W tym celu postępuj odwrotnie niż do tej pory, czyli wyodrębnij w kodzie funkcję anonimową, przypisz ją jakieś zmiennej, a następnie umieść tę zmienną tam, gdzie wcześniej było wyrażenie funkcyjne. Czy po takich zmianach kod jest bardziej czytelny? Co on robi?

```
var eat = function(food) {
    if (food === "ciasteczka") {
        alert("Poproszę o więcej.");
    } else if (food === "ciasto") {
        alert("Mniam mniam.");
    }
};
(eat)("ciasteczka");
```

Tutaj umieściliśmy wyodrębnioną funkcję. Nadaliśmy jej nazwę eat. Gdybyś wolał, mógłbyś ją także utworzyć przy użyciu deklaracji funkcji.

Oczywiście, tę instrukcję mógłbyś także zapisać eat("ciasteczka"), lecz chodziło nam o pokazanie, jak zastąpić wyrażenie funkcyjne zmienną eat.

A zatem w tym miejscu wywołujemy funkcję eat i przekazujemy do niej łańcuch znaków „ciasteczka". Ale do czego służą te dodatkowe nawiasy?

Sprawa wygląda tak. Czy pamiętasz, że deklaracje funkcji zaczynają się od słowa kluczowego function, po którym podawana jest nazwa funkcji? I czy pamiętasz, że wyrażenia funkcyjne muszą być podawane w jakiejś instrukcji? Jeśli zatem nie umieścimy nawiasów wokół wyrażenia funkcyjnego, interpreter JavaScriptu będzie chciał potraktować je jako deklarację, a nie jako wyrażenie funkcyjne. Jednak żeby wywołać funkcję eat, nawiasy nie są potrzebne; możemy je zatem usunąć.

A zatem cały ten kod służył tylko temu, by podać wyrażenie funkcyjne bezpośrednio w instrukcji, a następnie natychmiast wywołać tworzoną przez nie funkcję, przekazując do niej określony argument.

A... swoją drogą, ten kod wyświetla komunikat „Proszę o więcej.".

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Skorowidz

A
algebra Boole'a, 61
aplikacja, 343, 345

B
biblioteka, 74, 144, 506
  jQuery, *Patrz:* jQuery
błąd, 649
BOM, 647
Boole George, 61
Browser Object Model, *Patrz:* BOM

C
callback, *Patrz:* procedura obsługi zdarzeń
ciąg Fibonacciego, 654
closure, *Patrz:* domknięcie
CSS, 40, 48, 273, 345, 350
  reguła, 48, 284
  umiejscawianie, 354

D
Document Object Model, *Patrz:* DOM
dokument, 262, 264
DOM, 63, 64, 259, 263, 264, 267, 271, 274, 275, 278, 288, 289, 424, 646, 647
  element
    dodawanie, 288
    pobieranie, 288
    usuwanie, 288
domknięcie, 74, 124, 499, 517, 519, 520, 521, 526, 534
  tworzenie, 522, 525, 527
dziedziczenie, 584, 610, 628, 630
  prototypów, 589, 590, 591, 602
  przesłanianie, 628, 629
  w łańcuchu prototypów, 612, 621

E
element
  atrybut, 284, 285, 286
  body, 42, 70, 278, 346
  div, 347, 350
  docelowy, 424
  dodawanie, 288
  form, 354
  head, 70, 278
  pobieranie, 288
  script, 42, 68, 70, 73
  td, 347, 354
  usuwanie, 288
  zmiana zawartości, 272

F
Fibonacciego ciąg, 654
free variable, *Patrz:* zmienna niezależna
function expression, *Patrz:* wyrażenie funkcyjne
funkcja, 115, 119, 124, 125, 126, 133, 147, 150, 190, 238, 453, 465, *Patrz też:* metoda
  alert, 63, 64, 84, 91, 247, 647
  anonimowa, 124, 238, 499, 500, 501, 506, 534
  argument, 121, 124, 125, 126, 150
    przekazywany przez wartość, 128, 129, 224
  close, 647
  confirm, 647
  console.log, 63, 64, 65, 67, 106, 217, 247, 628
  definiowanie, 507, 508, 510, 518
  deklaracja, 145, 454, 455, 456, 460, 461, 465, 488, 500, 508, 510, 534
    zmiennej, 133
  document.write, 63, 64
  domykanie, 520
  dostępna w obiekcie, *Patrz:* metoda
  jako wynik wykonania innej funkcji, 474
  konstruktora, 548, 556
  Math.floor, 106, 109
  Math.PI, 573
  Math.random, 105, 109, 124, 391, 573
  nazwa, 133, 461
  o zmiennej liczbie argumentów, 648
  parametr, 124, 125, 126, 129, 150, 233, 234
    undefined, 130
  pierwszorzędna, 74
  porównująca, 483, 485
  prompt, 84, 91, 93, 109, 647
  przekazywanie do funkcji, 471, 472, 479
  push, 185, 200
  referencja, *Patrz:* referencja do funkcji
  setInterval, 436, 439, 444, 450, 647
  setTimeout, 433, 434, 435, 437, 439, 444, 647, 503
  sortująca, 465, 482
  sparametryzowana, 120, 124
  śledzenie przebiegu realizacji, 132
  środowisko, 515, 516, 518, 526
  timerHandler, 433, 647
  wbudowana, 106, 109, 150
  właściwość, 595
  wywoływanie, 121, 462, 463
  zagnieżdżanie, 509, 510, 512, 518, 534
  zasięg
    globalny, 508, 510
    leksykalny, 512, 513, 518, 534
    lokalny, 510

H
hermetyzacja, 217, 232, 249, 355
hoisting, *Patrz:* zmienna wyciąganie
HTML, 40, 48, 87, 260, 263, 264, 267, 345, 346, 354
  tabela, 348

I
instrukcja, 48, 51
  if, 60
  ifelse, 67
  pętli, *Patrz:* pętla
  return, 131, 132, 133, 150
  try/catch, 649
  warunkowa, 458

Kup książkę
Poleć książkę

interfejs programistyczny, 63
Internet Explorer, 438, 425
obsługa zdarzeń, 651

J

JavaScript, 40, 263, 267
biblioteka, *Patrz:* biblioteka
historia, 44
instrukcja, *Patrz:* instrukcja
kod, 41, 42, 71, 147, 278, 279
asynchroniczny, 413
blok, 61
dodawanie, 70
po stronie serwera, 657
pseudokod, *Patrz:* pseudokod
wyrażenie, *Patrz:* wyrażenie
zmienna, *Patrz:* zmienna
JavaScript Object Notation, *Patrz:* JSON
język
HTML, *Patrz:* HTML
interpretowany, 43
JavaScript, *Patrz:* JavaScript
kompilowany, 43, 74
skryptowy, 43, 74
jQuery, 644
JSON, 656

K

klasa, 584
kompilacja, 43
komunikat, 63
konsola, 63, 65, 106
komunikat o błędzie, 649
otwieranie, 66
konstruktor obiektów, 324, 543, 547, 550, 558, 569, 570, 575, 586
Array, 570, 573
Date, 570, 573
Error, 573
Math, 573
metoda, 552
nazwa, 554
Object, 573
parametr, 554
RegExp, 573, 652
stosowanie, 549, 555, 556, 558, 626
tworzenie, 548, 552
właściwość, 595

L

liczba losowa, *Patrz:* zmienna wartość losowa
literał
obiektowy, 543, 544, 554, 556, 561, 562, 568, 573, 575, 626
tablicowy, 185, 186, 200, 572

Ł

łańcuch odwołań, 371, 374, 399
łańcuch prototypów, *Patrz:* prototyp łańcuch
łańcuch znaków, 65, 93, 96, 303, 322, 323, 324, 334
jako obiekt, 323, 324
konkatenacja, 314
konwersja, 93, 304, 305, 309, 314, 315
length, 325
metoda
charAt, 325
concat, 328
indexOf, 326
lastIndexOf, 328
match, 328
replace, 328
slice, 328
split, 327
substring, 327
toLowerCase, 328
toUpperCase, 327
porównywanie, 309
pusty, 306, 319, 320

M

metoda, 230, 232, 233, 238, 483, *Patrz też:* funkcja
addEventListener, 650
charAt, 325, 329
concat, 328
createServer, 657
every, 571
getAttribute, 417
getElementById, 267, 270, 275, 286, 289, 298, 415, 417, 647
getElementsByClassName, 275, 646
getElementsByName, 646
getElementsByTagName, 422, 423, 444
hasOwnProperty, 606, 627
history, 647
indexOf, 326, 369, 399
innerHeight, 647
innerWidth, 647
join, 571
lastIndexOf, 328
location, 647
match, 328
print, 647
querySelector, 646
querySelectorAll, 646
replace, 328
reverse, 571
setAttribute, 285, 289, 417
slice, 328
sort, 483, 488
split, 327
substring, 323, 327, 328, 330, 630
toLowerCase, 328
toString, 627, 628
toUpperCase, 327
model obiektowy, 41
bazujący na klasach, 583
bazujący na prototypach, 583, 588
dokumentu, *Patrz:* DOM
przeglądarki, *Patrz:* BOM

N

NaN, 300, 301, 302, 309, 319, 320, 334
Node.js, 657
null, 91, 93, 275, 278, 286, 298, 299, 302, 306, 319, 320, 334

O

obiekt, 124, 129, 144, 205, 206, 211, 217, 246, 247, 294, 322, 333, 334, 543
arguments, 648
Array, 570
błędu, *Patrz:* obiekt Error
Date, 570
diagram, 585
document, 261
elementu, 261, 275
Error, 573
instancja, 565, 606, 626
konstruktor, *Patrz:* konstruktor obiektów
kontroler, 355, 375, 381, 399
Math, 573

Kup książkę Poleć książkę

modelu, 355, 362, 367, 399
myarray, 186
NodeList, 646
Object, 573, 627, 629
porównywanie, 316, 317
przekazywanie do funkcji, 224
referencja, 218, 219, 249, 317, 550, 551
RegExp, 573, 653
String, 631
tworzenie, 209, 543, 544, 568, 573, 592
w oparciu o konwencję, 545
wbudowany, 249, 294, 573, 629, 630, 631
widok, 355, 356, 357, 358, 399
window, 279, 647
właściwość, *Patrz:* właściwość
zachowanie, 206, 230, 242, 249
zdarzenia, 424, 425, 439
obsługa wyjątków, 649
operator
--, 180, 200
!, *Patrz:* operator NOT
!=, 93
!==, 309
&&, *Patrz:* operator AND
||, *Patrz:* operator OR
+, 96, 314, 315
++, 180, 200
<, 93
<=, 93, 309
=, 54, 313
==, 54, 93, 303, 304, 308, 309, 310, 312, 313, 317, 334
===, 93, 308, 309, 310, 312, 313, 317, 334
====, 313
>, 93
>=, 93, 309
alternatywy logicznej, *Patrz:* operator OR
AND, 93, 573
boolowski, 93
identyczności, *Patrz:* ===
instanceof, 333, 565, 573, 575, 621
logiczny, 93, 100, 109
new, 548, 551, 554, 555, 556, 557, 565, 575
NOT, 93
OR, 93, 94, 95
porównania, 93, 109
porównania, *Patrz:* ==
postdekrementacji, *Patrz:* operator --
postinkrementacji, *Patrz:* operator ++
równości, *Patrz:* ==
ścisły, *Patrz:* ===
typeof, 297, 333, 564

P

pętla, 89, 90
do while, 390, 399
for, 55, 174, 178, 181, 186, 200
for in, 55, 241
forEach, 55
nieskończona, 93
while, 55, 56, 67, 172, 178, 200
podnoszenie, *Patrz:* zmienna wyciąganie
procedura obsługi zdarzeń, 280, 282, 289, 385, 399, 419, 422, 444
addEventListener, 650
domknięcie, 527
tworzenie, 411, 416, 500, 527
programowanie
asynchroniczne, 407, 439
obiektowe, 212, 247
prototyp, 589, 601, 602, 610, 626
łańcuch, 610, 611, 626, 627
dziedziczenie, 612, 621
Object, 627
przesłanianie, 593, 602, 629
tworzenie, 596
przetwarzanie w chmurze, 657
pseudokod, 85, 88, 109
implementacja, 89

Q

QA, 99
Quality Assurance, *Patrz:* QA

R

refaktoryzacja, 190, 193
referencja
do funkcji, 435, 454, 463, 465, 488, 501, 503, 505, 525, 534
do metody, 586
do obiektu, *Patrz:* obiekt referencja
rekurencja, 654

S

schemat blokowy, 83
selektor, 646
słowo kluczowe, 50
break, 50
case, 50
catch, 50
class, 50
const, 50
continue, 50
debugger, 50
default, 50
delete, 50, 216
do, 50
else, 50
enum, 50
export, 50
extends, 50
false, 50
finally, 50
for, 50
if, 50
implements, 50
import, 50
in, 50
instanceof, 50, *Patrz też:* operator instanceof
interface, 50
let, 50
new, 50, 549, 551, 554, 555, 667, 565
package, 50
private, 50
protected, 50
public, 50
return, 50
static, 50
super, 50
switch, 50
this, 50, 234, 236, 237, 238, 249, 548, 550, 551, 554, 556, 600
throw, 50
true, 50
try, 50
typeof, 50, 297, 333, 564
var, 49, 50, 51, 133, 150

Kup książkę
Poleć książkę

słowo kluczowe
void, 50
while, 50
with, 50
yield, 50
sortowanie, 465, 481, 483, 485

T

tablica, 159, 161, 168, 184, 186, 200, 483, 648
element, 161, 200
pusta, 168, 185
równoległa, 196
rzadka, 185, 186
tworzenie, 162
wielkość, 164, 168
target, *Patrz:* element docelowy
typ, 293
Boolean, 61
konwersja, 305
dynamiczny, 74
konwersja, 294, 304, 305, 309, 314, 315
niskiego poziomu, 294
NodeList, 423, 646
null, 294, 299
prosty, 61, 219, 294, 322, 333, 334
undefined, 296, 297
wysokiego poziomu, 294

W

wartość
logiczna, 319, 320
losowa, *Patrz:* zmienna wartość losowa
pierwszej klasy, 466, 488, 505
windowanie, *Patrz:* zmienna wyciąganie
właściwość, 206, 209, 211, 213, 217, 234, 242
dodawanie, 214, 249
innerHTML, 260, 261, 269, 271, 272, 274, 275, 289
length, 325, 648
łańcucha znaków, *Patrz:* łańcuch znaków właściwość
onclick, 385, 415, 416, 417, 419, 421, 433
onload, 279, 411, 415, 433, 500, 501, 647
onmousemove, 431, 433
onmouseout, 440
onmouseover, 440
prototype, 595, 596
przeglądanie w pętli, 241
src, 417
target, 426, 427, 428
undefined, 296, 297, 319, 320, 334
usuwanie, 216, 249
wartość, 214
wyrażenie, 53, 61
funkcyjne, 454, 455, 458, 460, 461, 465, 488, 500, 501, 505, 510, 534
logiczne, 53, 60, 61, 100
regularne, 573, 652
warunkowe, 60, 92, 100, 103, 385
wywołanie
rekurencyjne, 654
zwrotne, *Patrz:* procedura obsługi zdarzeń
wzorzec, 543

Z

zapis z kropką, 213, 218, 219, 230, 249
zdarzenie, 282, 407, 409, 429, 439, 444
click, 416, 419, 422, 423, 425, 433, 445
dragstart, 445
drop, 445
keypress, 423
keypress, 445
kolejka, 429, 430
load, 410, 433, 445, 501
mousemove, 431, 433
mouseout, 440, 445
mouseover, 440, 445
obsługa w IE, 651
onload, 282, 289
pause, 445
play, 445
resize, 445
touchend, 445
touchstart, 445
unload, 445
zmienna, 49, 51, 88, 125
deklaracja, 139, 147
wewnątrz funkcji, 134
globalna, 134, 135, 138, 142, 144, 150, 233, 234
inicjalizowanie, 144
lokalna, 134, 135, 138, 139, 142, 144, 150, 233, 234, 238, 510, 513, 514, 520
deklaracja, 139, 144
nazwa, 50, 51, 136
niezależna, 519, 520, 525, 534
obiektowa, *Patrz:* zmienna referencyjna
przesłanianie, 140, 144
referencyjna, 218, 219
środowisko, 515, 520
tablicowa, 163
this, 550, 575, 600
wartość
Infinity, 302
losowa, 105, 109, 391
NaN, 300, 301, 302, 309, 319, 320, 334
null, *Patrz:* null
początkowa, 49, 51
undefined, 88, 144, 168, 186, 296, 306, 319, 334
wyciąganie, 144
zasięg, 134, 137, 144, 150
znacznik, *Patrz:* element
znak, *Patrz też:* opera

Kup książkę Poleć książkę

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

JavaScript dla każdego!

Programowanie w JavaScript. Rusz głową!

Jeżeli chcesz stworzyć atrakcyjną aplikację internetową lub stronę WWW, to poza znajomością języka HTML powinieneś również umieć korzystać z JavaScriptu. Język ten jeszcze całkiem niedawno wzbudzał wiele negatywnych emocji — ale te czasy odeszły w niepamięć! Bez jego pomocy współczesne strony WWW nie byłyby takie funkcjonalne!

Jeżeli chcesz poznać możliwości JavaScriptu oraz w pełni wykorzystać jego potencjał, trafiłeś na doskonałą książkę. Należy ona do cenionej serii „Rusz głową!" i opisuje wszystkie aspekty programowania w tym języku. Sięgnij po nią i poznaj język JavaScript od podstaw. Każda kolejna strona to spora dawka wiedzy podanej w przystępny sposób. Pomoże Ci poznać składnię języka, jego podstawowe elementy i konstrukcje. Kiedy opanujesz podstawy, przejdziesz do bardziej zaawansowanych tematów — programowania obiektowego, manipulowania drzewem DOM, obsługi zdarzeń oraz korzystania z funkcji anonimowych i domknięć. Książka ta jest doskonałą lekturą dla wszystkich osób chcących biegle władać językiem JavaScript!

Dzięki tej książce:

- poznasz składnię i podstawowe elementy języka JavaScript
- nauczysz się tworzyć obiekty i obsługiwać zdarzenia
- zobaczysz, jak wykorzystać domknięcia i funkcje anonimowe
- stworzysz prawdziwą grę

Nauka języka JavaScript jeszcze nigdy nie była tak przyjemna!

sięgnij po WIĘCEJ

KOD KORZYŚCI

helion.pl księgarnia internetowa

Nr katalogowy: 28688

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900 0 601 339900

Helion

Sprawdź najnowsze promocje: http://helion.pl/promocje

Książki najchętniej czytane: http://helion.pl/bestsellery

Zamów informacje o nowościach: http://helion.pl/nowosci

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

ISBN 978-83-246-9880-6

9 788324 698806

Cena 97,00 zł

Informatyka w najlepszym wydaniu