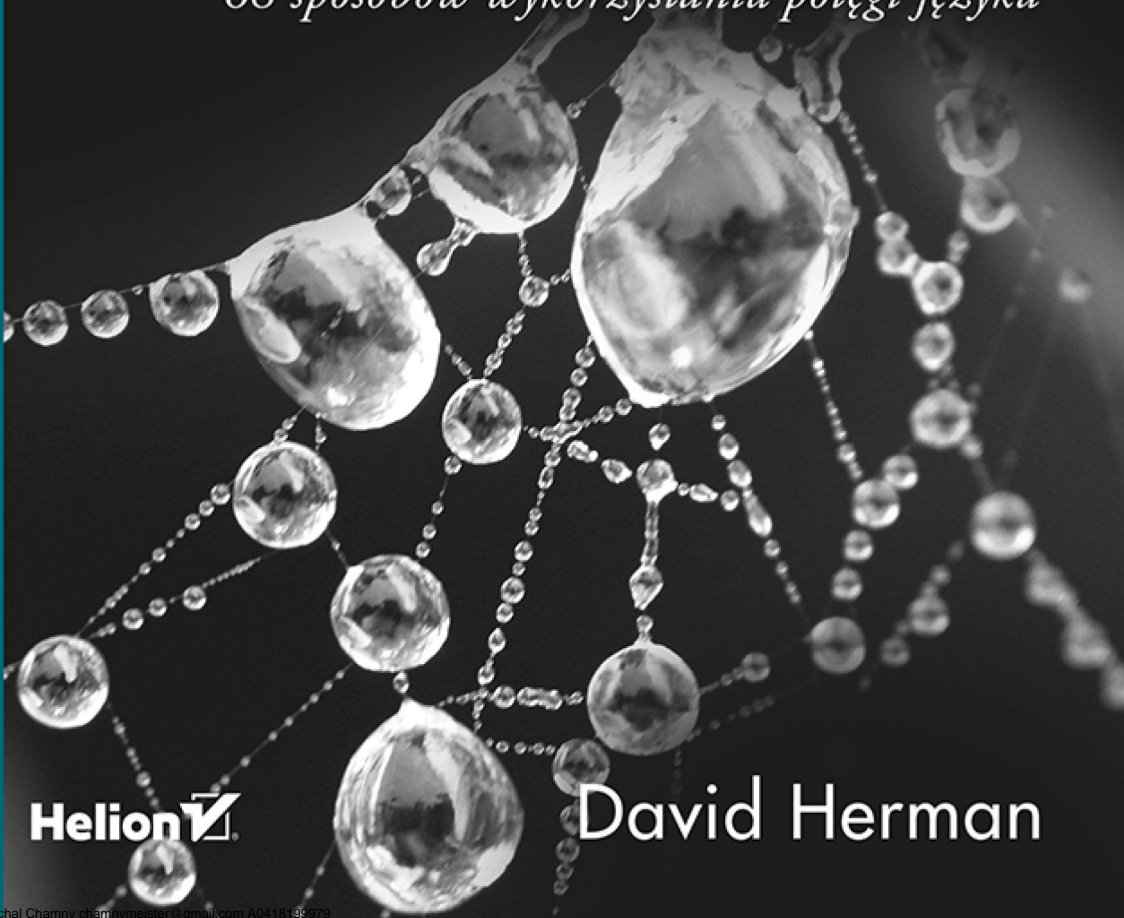




# *Efektywny* JAVASCRIPT

*68 sposobów wykorzystania potęgi języka*



**Helion**

David Herman

Tytuł oryginału: Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript  
Tłumaczenie: Tomasz Walczak  
ISBN: 978-83-283-1421-4

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

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

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

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

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

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

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

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[http://helion.pl/user/opinie/efprjs\\_ebook](http://helion.pl/user/opinie/efprjs_ebook)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

## Opinie na temat książki *Efektywny JavaScript. 68 sposobów wykorzystania potęgi języka*

---

„Książka *Efektywny JavaScript. 68 sposobów wykorzystania potęgi języka* Dave’a Hermana spełnia oczekiwania stawiane pozycjom z serii *Effective Software Development* i jest lekturą obowiązkową dla każdego, kto poważnie podchodzi do programowania w języku JavaScript. Znajdziesz tu szczegółowe wyjaśnienia wewnętrznych mechanizmów JavaScriptu, co pomoże Ci lepiej wykorzystać ten język”.

— Erik Arvidsson, starszy inżynier oprogramowania

„Nierzadko zdarza się spotkać specjalistę od języków programowania, który potrafi pisać tak zrozumiale i przystępnie, jak robi to David. Jego omówienie składni i semantyki JavaScriptu jest jednocześnie przyjemne w lekturze i bardzo wnikliwe. Wzmianki o kruczkach są uzupełnieniem realistycznych studiów przypadku, które pozwalają stopniowo pogłębiać wiedzę. Gdy skończysz lekturę tej książki, zauważysz w sobie mocne poczucie kompletnego opanowania tematu”.

— Paul Irish, Developer Advocate w firmie Google Chrome

„Przed lekturą książki *Efektywny JavaScript. 68 sposobów wykorzystania potęgi języka* myślałem, że będzie to jeszcze jedna pozycja o tym, jak pisać lepszy kod w JavaScriptcie. Jednak znajdziesz tu o wiele więcej; ta książka pozwoli Ci dogłębnie zrozumieć ten język. To niezwykle ważne. Bez takiego zrozumienia tak naprawdę w ogóle nie będziesz znał JavaScriptu. Będziesz jedynie wiedział, jaki kod piszą inni programiści.

„Jeśli chcesz być naprawdę dobrym programistą posługującym się JavaScriptem, przeczytaj tę książkę. Żałuję, że nie zrobiłem tego, gdy zaczynałem używać tego języka”.

— Anton Kovalyov, twórca wtyczki JSHint

„Jeśli szukasz książki, która zapewni Ci formalny, a jednocześnie bardzo przystępny wgląd w język JavaScript, to właśnie ją znalazłeś. Średnio zaawansowani programiści JavaScriptu znajdą tu prawdziwy skarb informacji, a nawet doświadczeni użytkownicy z pewnością nauczą się czegoś nowego. Dla zaawansowanych programistów innych języków chcących szybko poznać JavaScript ta książka jest lekturą obowiązkową, która pozwoli im szybko rozpocząć pracę. Niezależnie od poziomu doświadczenia czytelników Dave Herman wykonał fantastyczną pracę, opisując JavaScript — jego największe zalety, wady i wszystkie pośrednie cechy”.

— Rebecca Murphey,  
starszy programista JavaScriptu w firmie Bocoup

*Efektywny JavaScript. 68 sposobów wykorzystania potęgi języka* to niezbędna pozycja dla każdego, kto rozumie, że JavaScript to nie zabawka, i chce w pełni wykorzystać możliwości tego języka. Dave Herman oferuje czytelnikom dogłębne, sprawdzone i praktyczne zrozumienie języka. Przedstawia kolejne przykłady, aby odbiorcy mogli dojść do tych samych wniosków co autor. Nie jest to książka dla osób szukających dróg na skróty. Zamiast tego znajdziesz tu lata doświadczeń skondensowane do podróży z przewodnikiem. Jest to jedna z niewielu pozycji na temat JavaScriptu, którą mogę bez wahania polecić”.

— Alex Russell,  
członek grupy TC39, inżynier oprogramowania w firmie Google

„Rzadko mamy możliwość uczenia się od mistrzów w swoim fachu. Ta książka Ci ją zapewnia. Dzięki niej możesz w dziedzinie JavaScriptu poczuć się tak, jakbyś był podróżującym w czasie filozofem i mógł przenieść się w V wiek p.n.e., aby pobierać nauki u Platona”.

— Rick Waldron,  
osoba promująca JavaScript w firmie Bocoup

*Dla Lisy, mojej miłości.*



# Spis treści

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

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

### **Rozdział 3. Korzystanie z funkcji.....71**

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

### **Rozdział 4. Obiekty i prototypy .....95**

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



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

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

# Przedmowa

Jak już powszechnie wiadomo, utworzyłem JavaScript w dziesięć dni w maju 1995 roku pod presją i w obliczu sprzecznych instrukcji od zarządu: „niech przypomina Javę”, „ma być łatwy dla początkujących”, „ma zapewniać kontrolę niemal wszystkiego w przeglądarce Netscape”.

Zadbałem o właściwe rozwiązanie dwóch istotnych kwestii (udostępnienie prototypów obiektów i funkcji jako typów pierwszoklasowych), a oprócz tego w celu poradzenia sobie ze zmiennymi wymaganiami i absurdalnie krótkim terminem zdecydowałem się od samego początku umożliwić łatwe modyfikowanie JavaScriptu. Wiedziałem, że programiści będą poprawiać kilka pierwszych wersji języka, aby zlikwidować błędy i wprowadzić lepsze mechanizmy niż te, które udostępniłem w postaci bibliotek wbudowanych. W wielu językach możliwości wprowadzania zmian są niewielkie. Nie można na przykład modyfikować ani rozszerzać wbudowanych obiektów w czasie wykonywania kodu ani zmieniać powiązań z nazwami z biblioteki standardowej. Jednak JavaScript umożliwia niemal całkowitą modyfikację każdego obiektu.

Uważam, że była to właściwa decyzja projektowa. To prawda, stawia ona wyzwania w niektórych obszarach (na przykład przy bezpiecznym łączeniu zaufanego i niezaufanego kodu w granicach bezpieczeństwa w przeglądarce). Jednak konieczne było zapewnienie działania techniki monkey patching (polega ona na modyfikowaniu cudzego kodu) na potrzeby naprawiania błędów i dodawania obsługi przyszłych funkcji w starszych przeglądarkach (za pomocą bibliotek typu polyfill, zapewniających działanie technologii niedostępnych natywnie w przeglądarce).

Oprócz tych czasem przyziemnych zastosowań elastyczność JavaScriptu spowodowała powstawanie i rozrastanie się grup użytkowników wprowadzających innowacyjne rozwiązania. Najaktywniejsi wśród tych użytkowników tworzyli pakiety narzędzi lub wzorowane na innych językach biblioteki (Prototype na języku Ruby, MochiKit na Pythonie, Dojo na Javie, TIBET na Smalltalku).

Następnie pojawiła się biblioteka jQuery („nowa odsłona JavaScriptu”). Gdy w roku 2007 pierwszy raz się z nią zetknąłem, wydawała mi się spóźnionym produktem. Zyskała jednak bardzo dużą popularność w świecie JavaScriptu. Jej autorzy zrezygnowali z naśladowania innych języków i starszych bibliotek JavaScriptu, a w zamian dopracowali model „pobierz i przetwórz” stosowany w przeglądarce i znacznie go uprościli.

W efekcie dzięki najaktywniejszym użytkownikom i skupionym wokół nich grupom powstał charakterystyczny styl JavaScriptu, który do tej pory jest naśladowany i upraszczany w innych bibliotekach. Jest on też uwzględniany w pracach nad standaryzacją sieci WWW.

W trakcie ewolucji JavaScript pozostał zgodny wstecz (z uwzględnieniem błędów) i oczywiście modyfikowalny — nawet mimo wprowadzenia w najnowszej wersji standardu ECMAScript metod blokowania obiektów przed rozszerzaniem i zabezpieczania właściwości obiektów przed zmianami. Ewolucja JavaScriptu wciąż jest daleka od zakończenia. Podobnie jak w językach naturalnych i systemach biologicznych zmiana w długim okresie jest czymś stałym. Nie potrafię wyobrazić sobie jednej biblioteki standardowej ani stylu pisania kodu, który zastąpi wszystkie starsze podejścia.

Żaden język nie jest wolny od osobliwości. Języki nie są też na tyle restrykcyjne, aby wynikały z nich uniwersalne najlepsze praktyki. Także JavaScript nie jest pozbawiony osobliwości i nie jest restrykcyjny (a nawet wprost przeciwnie!). Dlatego aby skutecznie go używać, posługujący się nim programiści w większym stopniu niż użytkownicy innych języków powinni poznawać i stosować właściwy styl, zasady korzystania z niego oraz zalecane praktyki. Uważam przy tym, że przy szukaniu najskuteczniejszych rozwiązań bardzo ważne jest, aby unikać przesady i nie tworzyć sztywnych lub dogmatycznych wytycznych dotyczących stylu.

W tej książce zastosowano zrównoważone podejście, oparte na konkretnych dowodach i doświadczeniu. Autor unika sztywności i mnożenia reguł. Uważam, że ta pozycja będzie ważną pomocą i godnym zaufania podręcznikiem dla wielu osób, które chcą pisać efektywny kod w JavaScriptcie bez rezygnacji ze zwiezłości i swobody w wypróbowywaniu nowych pomysłów i paradygmatów. Ta książka jest konkretną i ciekawą lekturą pełną świetnych przykładów.

Mam przyjemność znać Davida Hermana od 2006 roku, kiedy to po raz pierwszy w imieniu organizacji Mozilla skontaktowałem się z nim, aby zaprosić go jako eksperta do jednostki pracującej nad standardami Ecma. Przekazywana w prosty sposób głęboka wiedza Dave’a i jego entuzjazm wobec JavaScriptu są widoczne na każdej stronie tej książki. Brawo!

— *Brendan Eich*

# Wprowadzenie

Nauka języka programowania wymaga poznania jego **składni**, czyli zestawu konstrukcji i struktur dozwolonych w poprawnych programach, oraz **semantyki**, czyli znaczenia lub działania tych konstrukcji. Ponadto aby opanować język, należy zrozumieć jego **pragmatyczne** aspekty, czyli sposób używania funkcji języka do tworzenia efektywnych programów. Ta ostatnia kwestia jest wyjątkowo skomplikowana — zwłaszcza w tak elastycznym i dającym dużą swobodę języku, jakim jest JavaScript.

Ta książka dotyczy praktycznych aspektów JavaScriptu. Nie jest to podręcznik dla początkujących. Zakładam, że masz już pewną wiedzę na temat JavaScriptu i programowania. Istnieje wiele doskonałych wprowadzeń do JavaScriptu, na przykład *JavaScript: The Good Parts* Douglasa Crockforda i *Eloquent JavaScript* Marijny Haverbeke’a. Moim celem jest sprawić, abyś lepiej zrozumiał, jak efektywnie używać JavaScriptu do pisania bardziej przewidywalnych, niezawodnych i łatwych w konserwacji aplikacji i bibliotek w JavaScriptcie.

## JavaScript a ECMAScript

Przed przejściem do materiału opisanego w tej książce warto doprecyzować terminologię. Ta książka jest poświęcona językowi powszechnie nazywanemu JavaScriptem. Jednak oficjalny standard opisujący specyfikację tego języka to ECMAScript. Historia tych nazw jest skomplikowana i związana z prawami autorskimi. Z powodów prawnych organizacja standaryzacyjna Ecma International nie mogła wykorzystać określenia JavaScript. Co więcej, organizacja ta musiała zmienić nazwę z pierwotnie używanej ECMA (jest to akronim od *European Computer Manufacturers Association*) na Ecma International, z małymi literami w pierwszym członie. Jednak do czasu dokonania tej zmiany utrzymała się już nazwa ECMAScript, z wielkimi literami.

Gdy ktoś używa określenia ECMAScript, zwykle ma na myśli idealny język opisany w standardzie organizacji Ecma. Natomiast nazwa JavaScript może oznaczać wiele rzeczy: od używanego w praktyce języka po konkretny silnik JavaScriptu rozwijany przez określonego producenta. Jednak często obie te nazwy są stosowane wymiennie. Aby zachować jednoznaczność i spójność, w tej książce używam określenia *ECMAScript* tylko do opisu oficjalnego standardu. W innych miejscach dla języka używam nazwy *JavaScript*. Ponadto posługuję się powszechnie używanym skrótem *ES5* do określania piątej wersji standardu ECMAScript.

## O sieci WWW

Trudno jest pisać o JavaScriptcie z pominięciem sieci WWW. Obecnie JavaScript to jedyny język programowania, którego obsługa jest wbudowana we wszystkich popularnych przeglądarkach internetowych, co pozwala na wykonywanie skryptów po stronie klienta. Ponadto dzięki pojawieniu się platformy Node.js w ostatnich latach JavaScript stał się popularnym językiem do tworzenia aplikacji działających po stronie serwera.

Jednak ta książka dotyczy JavaScriptu, a nie programowania rozwiązań dla sieci WWW. Czasem przedstawienie przykładów z obszaru sieci WWW i zastosowań pomysłów jest pomocne. Jednak książka poświęcona jest samemu językowi — składni, semantyce i aspektom pragmatycznym, a nie interfejsom API i technologiom związanym z siecią WWW.

## Uwagi na temat współbieżności

Ciekawą cechą JavaScriptu jest to, że jego działanie w środowisku współbieżnym jest zupełnie nieokreślone. Aż do piątej wersji włącznie w standardzie ECMAScript nie ma żadnych informacji na temat działania programów w JavaScriptcie w środowisku interaktywnym lub współbieżnym. Współbieżność jest omówiona w rozdziale 7. Znajdziesz tam techniczny opis nieoficjalnych funkcji JavaScriptu. W praktyce wszystkie popularne silniki JavaScriptu obsługują ten sam model współbieżności. Praca z programami współbieżnymi i interaktywnymi jest ważnym zagadnieniem zwiększającym jednolitość w programowaniu w JavaScriptcie, choć kwestie te nie są ujęte w standardzie. W przyszłych wersjach standardu ECMAScript wspólne aspekty modelu współbieżności z języka JavaScript mogą zostać oficjalnie sformalizowane.

# Podziękowania

Do powstania tej książki w dużym stopniu przyczynił się twórca JavaScriptu, Brendan Eich. Jestem mu bardzo wdzięczny za zaproszenie mnie do udziału w standaryzacji JavaScriptu oraz za jego nauki i wsparcie w czasie, gdy pracowałem w organizacji Mozilla.

W czasie pracy nad wieloma fragmentami tej książki czerpałem inspirację i wiedzę z doskonałych wpisów na blogach i artykułów z internetu. Oto osoby, z których wpisów wiele się dowiedziałem: Ben „cowboy” Alman, Erik Arvidsson, Mathias Bynens, Tim „creationix” Caswell, Michaeljohn „inimino” Clement, Angus Croll, Andrew Dupont, Ariya Hidayat, Steven Levithan, Pan Thomakos, Jeff Walden i Juriy „kangax” Zaytsev. Oczywiście ostatecznym źródłem informacji zawartych w tej książce jest specyfikacja ECMAScript, która od wersji piątej jest poprawiana i aktualizowana przez nieustrudzonego Allena Wirfsa-Brocka. Ponadto sieć Mozilla Developer Network wciąż jest jednym z najwyższej jakości internetowych zasobów dotyczących interfejsów API i specyfikacji JavaScriptu.

W trakcie projektowania i pisania tej książki miałem wielu doradców. John Resig udzielił mi przydatnych porad na temat pisania książek, zanim przystąpiłem do tego zadania. Blake Kaplan i Patrick Walton pomogli mi zebrać myśli i opracować układ tego podręcznika na wczesnych etapach prac. W trakcie pisania otrzymałem cenne wskazówki od Briana Andersona, Norberta Lindemberga, Sama Tobina-Hochstadta, Ricka Waldrona i Patricka Waltona.

Współpraca z osobami z wydawnictwa Pearson była prawdziwą przyjemnością. Olivia Basegio, Audrey Doyle, Trina MacDonald, Scott Meyers i Chris Zahn z uwagą odpowiadali na moje pytania, cierpliwie znosili spóźnienia i spełniali moje prośby. Nie potrafię wyobrazić sobie przyjemniejszych wrażeń z debiutu w roli autora. Jestem też zaszczycony, że mogłem włożyć swój wkład w rozwój serii Effective Software Development. Byłem fanem książki *C++. 50 efektywnych*

*sposobów na udoskonalenie Twoich programów* na długo przed tym, zanim mogłem choćby pomyśleć o tym, że będę miał przyjemność napisania książki z tej serii.

Miałem też wielkie szczęście, że redakcją techniczną książki zajęły się prawdziwe gwiazdy. Jestem zaszczycony tym, że Erik Arvidsson, Rebecca Murphey, Rick Waldron i Richard Worth zgodzili się redagować tę pozycję. Wskazali mi oni usterki i udzielili bezcennych wskazówek. Nieraz pomogli mi uniknąć naprawdę krępujących błędów.

Pisanie książki okazało się bardziej obciążające, niż sądziłem. Gdyby nie wsparcie przyjaciół i współpracowników, pewnie straciłbym nerwy. Andy Denmark, Rick Waldron i Travis Winfrey zachęcali mnie do pisania w momentach zwątpienia, choć pewnie nawet nie zdawali sobie z tego sprawy.

Zdecydowaną większość tej książki napisałem w cudownej kawiarni Java Beach Café w Parkside — pięknej dzielnicy San Francisco. Cały jej personel zna mnie z imienia i wie, co zamówię, zanim jeszcze zdążę to zrobić. Jestem im wdzięczny za to, że zapewnili mi przytulne miejsce pracy oraz jedzenie i porcję kafeiny.

Mój puszysty koci przyjaciel Schmoopy starał się, jak mógł, aby wnieść swój wkład w tę książkę. A przynajmniej wskakiwał na laptop i siadał przed ekranem. *Mogło* to mieć coś wspólnego z generowanym przez laptop ciepłem. Schmoopy jest moim lojalnym przyjacielem od 2006 roku. Nie potrafię wyobrazić sobie życia bez tego małego futrzaka.

Cała moja rodzina od początku do końca prac nad tym projektem była nim podekscytowana i mnie wspierała. Niestety, dziadek Frank Slamar i babcia Miriam Slamar odeszli, zanim zdążyłem pokazać im gotową książkę. Byli jednak podekscytowani i dumni ze mnie. W tej książce znalazło się krótkie wspomnienie z dzieciństwa dotyczące pisania razem z Frankiem programów w BASIC-u.

Miłości mojego życia, Lisie Silverii, zawdzięczam więcej, niż da się wyrazić we wprowadzeniu.



## O autorze

**David Herman** jest starszym pracownikiem badawczym w organizacji Mozilla Research. Posiada tytuł BA z dziedziny nauk komputerowych z uczelni Grinnell College oraz tytuły MS i PhD z tej samej dziedziny z uczelni Northeastern University. Jest też członkiem grupy Ecma TC39, odpowiedzialnej za tworzenie standardu języka JavaScript.



# 1

## Przyzwyczajanie się do JavaScriptu

JavaScript zaprojektowano tak, aby korzystanie z niego było intuicyjne. Dzięki składni podobnej jak w Javie i konstrukcjom występującym w wielu językach skryptowych (funkcjom, tablicom, słownikom i wyrażeniom regularnym) JavaScriptu szybko nauczy się każdy choć trochę doświadczony programista. Nawet początkujący po stosunkowo krótkiej nauce mogą szybko zacząć pisać programy, ponieważ język ten wymaga opanowania niewielkiej liczby podstawowych elementów.

Jednak choć JavaScript jest tak przystępny, dobre opanowanie go wymaga czasu i dogłębnego zrozumienia semantyki tego języka, jego osobliwości i najefektywniejszych idiomów. W każdym rozdziale tej książki omawiam inny obszar tematyczny związany z efektywnym programowaniem w JavaScriptcie. Niniejszy pierwszy rozdział rozpoczyna się od opisu najbardziej podstawowych zagadnień.

### Sposób 1. Ustal, której wersji JavaScriptu używasz

JavaScript, podobnie jak większość udanych technologii, ewoluuje w miarę upływu czasu. Pierwotnie miał być uzupełnieniem Javy stosowanym do programowania interaktywnych stron WWW. Ostatecznie jednak JavaScript zastąpił Javę jako główny język programowania w sieci WWW. Popularność JavaScriptu doprowadziła do jego sformalizowania w 1997 roku, kiedy to powstał międzynarodowy standard tego języka — ECMAScript. Obecnie istnieje wiele konkurencyjnych implementacji JavaScriptu, zgodnych z różnymi wersjami standardu ECMAScript.

Trzecia wersja standardu ECMAScript (ES3), ukończona w 1999 roku, wciąż jest najpowszechniejszą wersją JavaScriptu. Następnym ważnym usprawnieniem standardu była wersja piąta (ES5), udostępniona w roku 2009. W wersji ES5 wprowadzono liczne nowe funkcje, a ponadto ustandaryzowano niektóre powszechnie obsługiwane mechanizmy pominięte we wcześniejszych specyfikacjach. Ponieważ standard ES5 nie jest jeszcze powszechnie obsługiwany, w książce zwracam uwagę na zagadnienia lub wskazówki dotyczące właśnie tej wersji.

Oprócz wielu wersji standardu istnieją też liczne niestandardowe funkcje obsługiwane tylko w niektórych implementacjach JavaScriptu. Na przykład wiele silników JavaScriptu obsługuje słowo kluczowe `const` (używane przy definiowaniu zmiennych), jednak standard ECMAScript nie określa składni ani działania tego słowa. Ponadto działanie słowa kluczowego `const` jest różne w poszczególnych implementacjach. W niektórych z nich uniemożliwia ono modyfikowanie zmiennych:

```
constPI = 3.141592653589793;  
PI = "zmodyfikowana!";  
PI; // 3.141592653589793
```

W innych implementacjach `const` jest traktowane jak synonim słowa kluczowego `var`:

```
constPI = 3.141592653589793;  
PI = "zmodyfikowana!";  
PI; // Zmodyfikowana!
```

Z powodu długiej historii JavaScriptu i różnorodnych implementacji trudno jest stwierdzić, które funkcje są dostępne w poszczególnych platformach. Dodatkowym problemem jest to, że podstawowe środowisko JavaScriptu (czyli przeglądarki internetowe) nie daje programistom kontroli nad tym, która wersja JavaScriptu ma być używana do wykonywania kodu. Ponieważ użytkownicy końcowi mogą korzystać z różnych wersji rozmaitych przeglądarek, w trakcie pisania programów trzeba zachować ostrożność, aby działały one spójnie we wszystkich przeglądarkach.

JavaScript nie jest jednak używany tylko do pisania programów dla sieci WWW działających po stronie klienta. Służy też do pisania programów po stronie serwera, rozszerzeń przeglądarek oraz skryptów dla aplikacji mobilnych i desktopowych. W niektórych z tych sytuacji używana wersja JavaScriptu jest dokładnie znana. Wtedy warto wykorzystać dodatkowe funkcje dostępne w konkretnej implementacji JavaScriptu z danej platformy.

Ta książka dotyczy głównie standardowych funkcji JavaScriptu. Warto jednak omówić niektóre powszechnie obsługiwane, ale niestandardowe funkcje. Gdy używasz nowych standardów lub niestandardowych mechanizmów, koniecznie ustal, czy tworzone aplikacje będą działały w środowiskach obsłu-

gujących zastosowane rozwiązania. W przeciwnym razie może się okazać, że aplikacje będą działały w oczekiwany sposób na Twoim komputerze lub w systemie testowym, jednak zawiodą u użytkowników uruchamiających te programy w innych środowiskach. Na przykład słowo kluczowe `const` będzie działać prawidłowo w czasie testów w silniku, który obsługuje tę niestandardową funkcję, ale spowoduje błąd składni w przeglądarce internetowej nierozpoznającej tego słowa kluczowego.

W standardzie ES5 wprowadzono kolejny mechanizm związany z wersjami — **tryb *strict***. Pozwala on zastosować uproszczoną wersję JavaScriptu, w której niedostępne są problematyczne i narażone na błędy funkcje pełnego języka. Ten mechanizm zaprojektowano w taki sposób, aby był zgodny wstecz. Dzięki temu nawet w środowiskach, w których obsługa trybu *strict* nie jest zaimplementowana, można wykonywać kod napisany w tym trybie. Aby włączyć tryb *strict*, dodaj na samym początku programu specjalną stałą:

```
"use strict";
```

Tryb *strict* można też włączyć w funkcji. W tym celu umieść wspomnianą dyrektywę na początku ciała funkcji:

```
functionf(x) {  
    "use strict";  
    // ...  
}
```

Stosowanie dyrektywy w postaci literału znakowego może wydawać się dziwne, jednak zaletą tego rozwiązania jest zachowanie zgodności kodu wstecz. Przetwarzanie literału znakowego nie powoduje efektów ubocznych, dlatego w silniku ES3 ta dyrektywa jest tylko nieszkodliwą instrukcją. Silnik przetworzy łańcuch znaków, a następnie natychmiast pominie jego wartość. To pozwala pisać kod w trybie *strict* i uruchamiać go w starszych silnikach JavaScriptu. Obowiązuje przy tym ważne ograniczenie — starsze silniki nie sprawdzają, czy ustawiony jest tryb *strict*. Jeśli nie przetestujesz kodu w środowisku z silnikiem zgodnym ze standardem ES5, może się okazać, że w takim środowisku program nie zadziała:

```
functionf(x) {  
    "use strict";  
    var arguments = []; // Błąd: ponowna definicja zmiennej arguments  
    // ...  
}
```

W trybie *strict* ponowne definiowanie zmiennej `arguments` jest niedozwolone, jednak w środowisku, w którym ten tryb nie jest sprawdzany, przedstawiony kod zadziała. Jeśli potem udostępnisz ten kod, spowoduje on błędy w środowiskach z implementacją standardu ES5. Dlatego kod w trybie *strict* zawsze powinieneś testować w środowiskach w pełni zgodnych ze standardem ES5.

Jedną z pułapek związanych ze stosowaniem trybu *strict* jest to, że dyrektywa "use strict" działa tylko wtedy, gdy jest umieszczona na początku skryptu lub funkcji. Może to prowadzić do problemów przy **scalaniu skryptów**, kiedy to duże aplikacje są tworzone w odrębnych plikach, a następnie łączone w jeden plik stosowany w wersji produkcyjnej. Oto plik, który ma działać w trybie *strict*:

```
// file1.js
"use strict";
function f() {
    // ...
}
```

A oto inny plik, który nie musi działać w tym trybie:

```
// file2.js
// Nie ma tu dyrektywy trybu strict
function g() {
    var arguments = [];
    // ...
}
```

W jaki sposób poprawnie połączyć te dwa pliki? Jeśli zaczniesz od pliku *file1.js*, cały złączony plik będzie działał w trybie *strict*:

```
// file1.js
"use strict";
function f() {
    // ...
}
// ...
// file2.js
// Nie ma tu dyrektywy trybu strict
function f() {
    var arguments = []; // Błąd: ponowna definicja zmiennej arguments
    // ...
}
```

Jeśli natomiast zaczniesz od pliku *file2.js*, tryb *strict* w ogóle nie zostanie zastosowany:

```
// file2.js
// Nie ma tu dyrektywy trybu strict
function g() {
    var arguments = [];
    // ...
}
// ...
// file1.js
"use strict";
function f() { // Tryb strict nie obowiązuje
```

```
// ...
}
// ...
```

We własnych projektach możesz zastosować regułę „tylko tryb *strict*” lub „tylko bez trybu *strict*”. Jeśli jednak chcesz pisać niezawodny kod, który można łączyć z innymi plikami, warto poznać kilka innych rozwiązań.

*Rezygnacja z łączenia plików o różnych trybach.* Jest to prawdopodobnie najłatwiejsze podejście, jednak ogranicza zakres kontroli nad strukturą plików aplikacji lub biblioteki. W najlepszym razie trzeba zastosować dwa odrębne pliki — jeden łączący wszystkie pliki w trybie *strict* i drugi obejmujący wszystkie pliki bez trybu *strict*.

*Umieszczanie ciała złączanych plików w natychmiast wywoływanych wyrażeniach funkcyjnych.* Szczegółowe wyjaśnienie natychmiast wywoływanych wyrażań funkcyjnych (ang. *Immediately Invoked Function Expression* — IIFE) znajdziesz w sposobie 13. Dzięki umieszczeniu zawartości każdego pliku w funkcji można niezależnie interpretować kod w różnych trybach. Oto złączona wersja wcześniej przedstawionego przykładowego kodu:

```
// Bez dyrektywy trybu strict
(function() {
  // file1.js
  "use strict";
  function f() {
    // ...
  }
  // ...
})();

(function() {
  // file2.js
  // Bez dyrektywy trybu strict
  function f() {
    var arguments = [];
    // ...
  }
  // ...
})();
```

Ponieważ zawartość każdego pliku znajduje się w odrębnym zasięgu, dyrektywa trybu *strict* (lub jej brak) wpływa tylko na dany plik. Jednak aby to podejście zadziałało, nie można zakładać, że zawartość plików jest interpretowana w zasięgu globalnym. Na przykład deklaracje `var` i `function` nie zostaną utrwalone jako zmienne globalne (więcej informacji o elementach globalnych zawiera Sposób 8.). Taka sytuacja ma miejsce w popularnych *systemach modularnych*, w których w ramach zarządzania plikami i zależnościami zawartość każdego modułu jest umieszczana w odrębnej funkcji. Ponieważ wszystkie pliki są umieszczane w zasięgu lokalnym, dla każdego pliku można niezależnie ustawić, czy używany ma być tryb *strict*.

*Pisanie plików w taki sposób, aby niezależnie od trybu działały tak samo.* Jeśli chcesz napisać bibliotekę, która będzie działała w możliwie wielu kontekstach, nie możesz zakładać, że zostanie ona umieszczona w funkcji przez narzędzie scalające skrypty. Nie możesz też przyjmować, że w kodzie klienckim tryb *strict* będzie włączony (lub nie). Aby uzyskać maksymalną kompatybilność z innym kodem, najłatwiej jest używać trybu *strict* i bezpośrednio umieszczać kod w funkcjach. Pozwala to lokalnie włączać ten tryb. Przypomina to poprzednie rozwiązanie, w którym zawartość każdego pliku jest umieszczana w wyrażeniach IIFE, jednak tu takie wyrażenia są pisane ręcznie. Nie trzeba wtedy liczyć na to, że zrobią to narzędzia do scalania skryptów lub system modułowy. Poniżej jawnie ustawiany jest tryb *strict*:

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

Zauważ, że ten kod działa w trybie *strict* niezależnie od tego, czy zostanie scalony w kontekście z włączonym tym trybem. Natomiast funkcja, w której tryb *strict* nie jest włączony, też będzie działała w tym trybie, jeśli zostanie dołączona po kodzie z trybem *strict*. Dlatego bardziej uniwersalnym podejściem jest pisanie kodu w trybie *strict*.

## Co warto zapamiętać?

- Określ, z których wersji JavaScriptu ma korzystać rozwijana aplikacja.
- Upewnij się, że używasz funkcji JavaScriptu obsługiwanych we wszystkich środowiskach, w których aplikacja będzie uruchamiana.
- Kod w trybie *strict* zawsze testuj w środowiskach, które sprawdzają ten tryb.
- Unikaj scalania skryptów, w których oczekiwane są różne ustawienia trybu.

## Sposób 2. Liczby zmiennoprzecinkowe w JavaScriptcie

W większości języków programowania dostępnych jest kilka typów liczbowych. W JavaScriptcie używany jest tylko jeden taki typ. Jest to widoczne w działaniu operatora `typeof`, który informuje, że typem zarówno liczb całkowitych, jak i liczb zmiennoprzecinkowych jest `number`:



```
typeof 17;    // "number"
typeof 98.6;  // "number"
typeof -2.1;  // "number"
```

Wszystkie liczby w JavaScriptcie to wartości **zmiennoprzecinkowe o podwójnej precyzji**. Są to liczby kodowane za pomocą 64 bitów, opisane w standardzie IEEE 754, często występujące jako typ `double`. Co więc dzieje się z liczbami całkowitymi? Pamiętaj, że liczby o podwójnej precyzji mogą reprezentować liczby całkowite z precyzją do 53 bitów. Wszystkie liczby całkowite z przedziału od  $-9\,007\,199\,254\,740\,992$  ( $-2^{53}$ ) do  $9\,007\,199\,254\,740\,992$  ( $2^{53}$ ) to poprawne liczby o podwójnej precyzji. Dlatego mimo braku typu całkowitoliczbowego można w JavaScriptcie stosować arytmetykę całkowitoliczbową.

Większość operatorów arytmetycznych działa dla liczb całkowitych, liczb rzeczywistych i kombinacji wartości tych typów:

```
0.1 * 1.9 // 0.19
-99 + 100; // 1
21 - 12.3; // 8.7
2.5 / 5; // 0.5
21 % 8: // 5
```

W specjalny sposób działają operatory arytmetyki bitowej. Nie traktują one argumentów bezpośrednio jako liczb zmiennoprzecinkowych, ale niejawnie przekształcają je na 32-bitowe liczby całkowite (są one traktowane jak 32-bitowe liczby całkowite *w porządku big-endian z dopełnieniem do dwóch*). Oto przykładowe bitowe wyrażenie OR:

```
8 | 1; // 9
```

Przetworzenie tego na pozór prostego wyrażenia wymaga kilku kroków. W JavaScriptcie 8 i 1 to liczby o podwójnej precyzji. Można je jednak przedstawić jako 32-bitowe liczby całkowite, czyli sekwencję trzydziestu dwóch jedynek i zer. Wartość 8 zapisana jako 32-bitowa liczba całkowita wygląda tak:

[illegible]

Możesz się o tym przekonać samodzielnie. W tym celu zastosuj metodę `toString` z typu liczbowego:

```
(8).toString(2); // "1000"
```

Argument metody `toString` określa **podstawę**. Tu używana jest reprezentacja o podstawie 2 (czyli reprezentacja dwójkowa). W wyniku dodatkowe bity 0 występujące po lewej stronie są pomijane, ponieważ nie wpływają na wartość liczby.

Liczba całkowita 1 jest reprezentowana za pomocą 32 bitów w następujący sposób:

00000000000000000000000000000000000001

[illegible]

```
parseInt("1001",2); // 9
```

```
0.1 + 0.2: // 0.30000000000000004
```

```
(0.1 + 0.2) + 0.3; // 0.6000000000000001
0.1 + (0.2 + 0.3); // 0.6
```

helion kopia dla: Michal Chamny chamnymeister@gmail.com A0418199979

takich liczb. Dobrym rozwiązaniem jest stosowanie wszędzie tam, gdzie to możliwe, liczb całkowitych, ponieważ można je przedstawiać bez zaokrąglania. Przy obliczeniach finansowych programiści często zmieniają skalę i jako jednostkę stosują najmniejszy nominal, co pozwala używać liczb całkowitych. Na przykład gdyby wcześniej przedstawione obliczenia były wykonywane w złotych, można przekształcić je na operacje na groszach:

```
(10 + 20) + 30; // 60
10 + (20 + 30); // 60
```

Przy stosowaniu liczb całkowitych nadal trzeba uważać, aby wszystkie obliczenia mieściły się w przedziale od  $-2^{53}$  do  $2^{53}$ , ale nie trzeba się wtedy przejmować błędami zaokrąglania.

### Co warto zapamiętać?

- Liczby w JavaScriptcie to wartości zmiennoprzecinkowe o podwójnej precyzji.
- Liczby całkowite w JavaScriptcie to podzbiór liczb o podwójnej precyzji, a nie odrębny typ danych.
- Operatory bitowe interpretują liczby jako 32-bitowe liczby całkowite ze znakiem.
- Pamiętaj o ograniczeniach precyzji w arytmetyce zmiennoprzecinkowej.

### Sposób 3. Uważaj na niejawną konwersję typu

JavaScript bywa zaskakująco pobłażliwy, jeśli chodzi o błędy typów. W wielu językach wyrażenia takie jak poniższe powodują błąd:

```
3 + true; // 4
```

Jest tak, ponieważ wyrażenia logiczne (takie jak `true`) są niekompatybilne z operacjami arytmetycznymi. W języku ze statyczną kontrolą typów programu z takim wyrażeniem nie można nawet uruchomić. W niektórych językach z dynamiczną kontrolą typów taki program rozpocznie pracę, jednak pokazane wyrażenie spowoduje wyjątek. JavaScript nie tylko uruchomi program z przedstawionym wyrażeniem, ale też szybko zwróci wynik 4!

W niektórych sytuacjach w JavaScriptcie użycie niewłaściwego typu powoduje natychmiastowe zwrócenie błędu. Dzieje się tak na przykład przy wywołaniu czegoś, co nie jest funkcją, lub przy próbie pobrania właściwości wartości `null`:

```
"hello"(1); // Błąd: to nie funkcja
null.x;      // Błąd: nie można wczytać właściwości „x” wartości null
```

Jednak w wielu innych sytuacjach zamiast zgłaszać błąd, JavaScript dokonuje **konwersji typu** wartości na oczekiwany typ, posługując się różnymi protokołami automatycznej zmiany typów. Na przykład operatory arytmetyczne `-`, `*`, `/` i `%` przed przeprowadzeniem obliczeń próbują przekształcić wszystkie argumenty na liczby. Operator `+` działa bardziej subtelnie, ponieważ jest przeciążony i w zależności od typów argumentów dodaje liczby lub scala łańcuchy znaków:

```
2 + 3; // 5
"witaj," + " świecie"; // "witaj, świecie"
```

Co się jednak stanie, jeśli spróbujesz dodać liczbę i łańcuch znaków? JavaScript faworyzuje wtedy łańcuchy znaków i przekształca liczbę na postać tekstową:

```
"2" + 3; // "23"
2 + "3"; // "23"
```

Łączenie typów w ten sposób bywa źródłem problemów, zwłaszcza że istotna jest przy tym kolejność operacji. Przyjrzyj się poniższemu wyrażeniu:

```
1 + 2 + "3"; // "33"
```

Ponieważ przy dodawaniu wartości są grupowane od lewej strony (jest to operacja **łączna lewostronnie**), ta instrukcja zadziała tak samo jak poniższa:

```
(1 + 2) + "3"; // "33"
```

Poniższe wyrażenie zadziała inaczej:

```
1 + "2" + 3; // "123"
```

To wyrażenie zwróci łańcuch znaków „123”. Z powodu łączności lewostronnej ten kod działa tak samo jak wyrażenie z pierwszą operacją dodawania umieszczoną w nawiasie:

```
(1 + "2") + 3; // "123"
```

Operacje bitowe nie tylko przekształcają wartości na liczby, ale też używają określonego podzbioru liczb — 32-bitowych liczb całkowitych, co opisano w sposobie 2. Dotyczy to bitowych operatorów arytmetycznych (`~`, `&`, `^` i `|`) i operatorów przesunięcia (`<<`, `>>` i `>>>`).

Niejawna konwersja typów bywa kusząco wygodna. Na przykład pozwala automatycznie przekształcać łańcuchy znaków wprowadzane przez użytkowników, pobierane z pliku tekstowego lub ze strumienia danych z sieci:

```
"17" * 3; // 51
"8" | "1"; // 9
```

Jednak taka konwersja może też prowadzić do ukrywania błędów. Zmienna o wartości `null` nie spowoduje błędu w obliczeniach arytmetycznych, ale zostanie niejawnie przekształcona w wartość `0`. Zmienna niezdefiniowana

(undefined) zostanie przekształcona w specjalną wartość zmiennoprzecinkową NaN (ang. *not a number*, czyli „nieliczba”, paradoksalnie mamy więc liczbę o nazwie „nieliczba”; winny temu jest standard IEEE opisujący liczby zmiennoprzecinkowe). Niejawna konwersja typów ukrywa zatem błąd i powoduje kontynuowanie obliczeń z często mylącymi i nieprzewidywalnymi wynikami. Frustrujące jest to, że trudno nawet sprawdzić, czy dana wartość to NaN. Wynika to z dwóch powodów. Po pierwsze, JavaScript jest zgodny ze zdumiewającym wymogiem ze standardu IEEE opisującego liczby zmiennoprzecinkowe, według którego wartość NaN ma być traktowana jako niezgodna z samą sobą. Dlatego sprawdzanie, czy wartość jest równa NaN, nie zadziała:

```
var x = NaN;
x === NaN; // false
```

Ponadto standardowa funkcja biblioteczna `isNaN` bywa zawodna, ponieważ sama stosuje niejawną konwersję typu i przekształca argument na liczbę przed sprawdzeniem jego wartości (dlatego lepszą nazwą dla `isNaN` byłoby `zmieniaWNaN`). Jeśli już wiesz, że dana wartość to liczba, możesz za pomocą funkcji `isNaN` sprawdzić, czy jest równa NaN:

```
isNaN(NaN); // true
```

Jednak inne wartości, które z pewnością są różne od NaN, ale zostają przekształcone w tę wartość, są przez funkcję `isNaN` traktowane w identyczny sposób:

```
isNaN("foo"); // true
isNaN(undefined); // true
isNaN({}); // true
isNaN({ valueOf: "foo" }); // true
```

Na szczęście istnieje idiom, który pozwala w niezawodny i zwięzły (choć nieco nieintuicyjny) sposób sprawdzić, czy dana wartość jest równa NaN. Ponieważ NaN to jedyna wartość w JavaScriptcie, która jest traktowana jako różna od siebie samej, można wykrywać ją, sprawdzając, czy dana wartość jest sobie równa:

```
var a = NaN;
a !== a; // true
var b = "foo";
b !== b; // false
var c = undefined;
c !== c; // false
var d = {};
d !== d; // false
var e = { valueOf: "foo" };
e !== e; // false
```

Ten wzorzec można zapisać w postaci odpowiednio nazwanej funkcji narzędziowej:

```
function isReallyNaN(x) {  
    return x !== x;  
}
```

Jednak testowanie nierówności wartości względem niej samej jest na tyle zwięzłe, że często stosuje się to rozwiązanie bez funkcji pomocniczej. Warto zatem znać i rozumieć to rozwiązanie.

Niejawna konwersja ukrywa błędy i utrudnia ich diagnozę, co sprawia, że debugowanie niedziałających programów bywa frustrujące. Gdy obliczenia zwracają niewłaściwy wynik, najlepszą metodą debugowania jest sprawdzenie pośrednich wyników w celu dojścia do ostatniego punktu sprzed wystąpienia problemów. Następnie należy zbadać argumenty każdej operacji i poszukać tych o niewłaściwym typie. Czasem przyczyną może być błąd logiczny, wynikający na przykład z zastosowania niewłaściwego operatora arytmetycznego, lub błąd typu, taki jak przekazanie wartości `undefined` zamiast liczby.

Ponadto obiekty mogą być przekształcane na typy proste. Ta technika najczęściej służy do przekształcania na łańcuchy znaków:

```
"Obiekt Math: " + Math; // "Obiekt Math: [object Math]"  
"Obiekt JSON: " + JSON; // "Obiekt JSON: [object JSON]"
```

Obiekty są przekształcane na łańcuchy znaków w wyniku niejawnego wywołania metody `toString`. Aby to sprawdzić, możesz sam wywołać tę metodę:

```
Math.toString(); // "[object Math]"  
JSON.toString(); // "[object JSON]"
```

Ponadto obiekty mogą być przekształcane na liczby za pomocą metody `valueOf`. Aby kontrolować konwersję obiektów, odpowiednio zdefiniuj te metody:

```
"J" + { toString: function() { return "S"; } }; // "JS"  
2 * { valueOf: function() { return 3; } }; // 6
```

Sytuację komplikuje to, że operator `+` jest przeciążony i potrafi zarówno scalać łańcuchy znaków, jak i dodawać liczby. Gdy obiekt udostępnia metody `toString` i `valueOf`, nie jest oczywiste, którą z nich operator `+` powinien wywołać. Ten operator powinien wybrać scalanie lub dodawanie na podstawie typów, jednak przy niejawnym konwersji typów nie są one określone! Rozwiązanie tej niejednoznaczności w JavaScriptcie polega na domyślnym stosowaniu metody `valueOf` zamiast `toString`. To jednak sprawia, że jeśli użytkownik chce scalać łańcuch znaków z obiektem, może otrzymać nieoczekiwane wyniki:

```
var obj = {  
    toString: function() {  
        return "[object MyObject]";  
    },  
    valueOf: function() {  
        return 17;  
    }  
}
```

```
};
"Obiekt: " + obj; // "Obiekt: 17"
```

Morał z tego jest taki, że metodę `valueOf` zaprojektowano do użytku w obiektach reprezentujących wartości liczbowe (na przykład w obiektach typu `Number`). Dla takich obiektów metody `toString` i `valueOf` zwracają spójne wyniki — tekstową lub liczbową reprezentację tej samej wartości. Dlatego przeciążony operator `+` działa spójnie niezależnie od tego, czy obiekt jest w operacji skalania, czy dodawania. Przeważnie niejawna konwersja na łańcuchy znaków jest przydatniejsza i częściej stosowana niż konwersja na liczby. Najlepiej jest unikać metody `valueOf`, chyba że obiekt reprezentuje liczbę, a metoda `obj.toString()` zwraca tekstową reprezentację wartości wywołania `obj.valueOf()`.

Ostatni rodzaj niejawnej konwersji typów jest związany z **prawdziwością**. Operatory `if`, `||` i `&&` używają wartości logicznych, jednak akceptują dowolne dane. W JavaScriptcie dane są interpretowane jako wartości logiczne zgodnie z prostą niejawną konwersją typu. Wartości w JavaScriptcie w większości są traktowane jako *prawdziwe*, czyli są niejawnie przekształcane na wartość `true`. Dotyczy to także obiektów. Inaczej niż przy konwersji na łańcuchy znaków i liczby tu nie jest niejawnie wywoływana żadna metoda. Istnieje siedem wartości oznaczających *fałsz*: `false`, `0`, `-0`, `""`, `NaN`, `null` i `undefined`. Wszystkie pozostałe wartości są traktowane jako prawdziwe. Ponieważ liczby i łańcuchy znaków mogą być traktowane jak fałsz, niebezpiecznie jest sprawdzać na podstawie prawdziwości, czy określone argumenty funkcji lub właściwości obiektu są zdefiniowane. Przyjrzyj się funkcji, która przyjmuje opcjonalne argumenty o wartościach domyślnych:

```
function point(x, y) {
  if (!x) {
    x = 320;
  }
  if (!y) {
    y = 240;
  }
  return { x: x, y: y };
}
```

Ta funkcja ignoruje wszystkie argumenty odpowiadające wartości fałsz, w tym wartości `0`:

```
point(0, 0); // {x: 320, y: 240}
```

Bardziej precyzyjną metodą wykrywania wartości `undefined` jest zastosowanie operatora `typeof`:

```
function point(x, y) {
  if (typeof x === "undefined") {
    x = 320;
  }
  if (typeof y === "undefined") {
    y = 240;
  }
}
```

```
    return { x: x, y: y };  
}
```

Ta wersja funkcji `point` poprawnie odróżnia 0 od wartości `undefined`:

```
point();           // {x: 320, y: 240}  
point(0, 0);      // {x: 0, y: 0}
```

Inna technika polega na porównaniu zmiennej z wartością `undefined`:

```
if (x === undefined) { ... }
```

W sposobie 54. opisuję wpływ sposobu sprawdzania prawdziwości na projekty bibliotek i interfejsów API.

### Co warto zapamiętać?

- Z powodu niejawnej konwersji typów błędy typów mogą być niezauważalne.
- Operator `+` jest przeciążony i w zależności od typów argumentów dodaje liczby lub scala łańcuchy znaków.
- Obiekty są przekształcane na liczby za pomocą metody `valueOf` i na łańcuchy znaków za pomocą metody `toString`.
- W obiektach z metodą `valueOf` należy zaimplementować metodę `toString`, która powinna zwracać tekstową reprezentację liczby zwracanej przez metodę `valueOf`.
- Do wykrywania niezdefiniowanych wartości używaj operatora `typeof` lub porównania z wartością `undefined`.

### Sposób 4. Stosuj typy proste zamiast nakładek obiektowych

W JavaScriptcie oprócz obiektów występuje pięć rodzajów typów prostych: typy logiczne, liczby, łańcuchy znaków, `null` i `undefined`. Mylące jest to, że operator `typeof` zwraca dla wartości `null` komunikat "object", podczas gdy w standardzie ECMAScript `null` jest opisane jako odrębny typ. W bibliotece standardowej dostępne są konstruktory do tworzenia nakładek obiektowych na typy logiczne, liczby i łańcuchy znaków. Możesz na przykład użyć obiektu `String` obejmującego wartość tekstową:

```
var s = new String("Witaj.");
```

Pod niektórymi względami obiekt typu `String` działa podobnie jak umieszczony w nim łańcuch znaków. Możesz na przykład scalić taki obiekt z inną wartością, aby uzyskać łańcuch znaków:

```
s + " świecie"; // "Witaj, świecie"
```



Możesz też pobrać podłańcuch o określonym indeksie:

```
s[4]; // "o"
```

Jednak w odróżnieniu od prostych łańcuchów znaków obiekt typu `String` jest prawdziwym obiektem:

```
typeof "Witaj."; // "string"  
typeof s;        // "object"
```

To ważna różnica, ponieważ oznacza, że nie można porównywać zawartości dwóch różnych obiektów typu `String` za pomocą wbudowanych operatorów:

```
var s1 = new String("Witaj.");  
var s2 = new String("Witaj.");  
s1 === s2; // false
```

Ponieważ każdy obiekt typu `String` jest odrębnym obiektem, jest identyczny tylko z sobą samym. To samo dotyczy operatora równości `==`:

```
s1 == s2; // false
```

Ponieważ nakładki nie działają właściwie, nie są zbyt przydatne. Głównym uzasadnieniem ich istnienia są metody narzędziowe. JavaScript umożliwia wygodne korzystanie z nich dzięki niejawnej konwersji typów. Możesz stosować właściwości i metody dla wartości typu prostego, a zadziała ona, jakby była umieszczona w odpowiednim typie obiektowym. Na przykład obiekt typu `String` udostępnia metodę `toUpperCase`, która przekształca litery w łańcuchu znaków na wielkie. Tę metodę możesz zastosować dla zwykłego łańcucha znaków:

```
"Witaj.".toUpperCase(); // "WITAJ,"
```

Dziwną konsekwencją tego niejawnego stosowania nakładek jest to, że można ustawiać właściwości typów prostych, co jednak nie daje żadnych trwałych efektów:

```
"Witaj.".someProperty = 17;  
"Witaj.".someProperty; // undefined
```

Ponieważ niejawne stosowanie nakładek powoduje utworzenie za każdym razem nowego obiektu typu `String`, modyfikacja pierwszego obiektu nie powoduje trwałych skutków. Dlatego ustawianie właściwości dla prostych wartości nie ma sensu, jednak warto wiedzieć o tej technice. Jest to jedno z miejsc, w których JavaScript ukrywa błędy typów — jeśli ustawiasz właściwości dla czegoś, co uważasz za obiekt, ale pomyłkowo zastosujesz wartość typu prostego, program zignoruje modyfikacje i będzie kontynuował pracę. Może to spowodować, że błąd pozostanie niezauważony, i utrudni diagnozę problemu.

## Co warto zapamiętać?

- Porównywanie dla nakładek obiektowych na typy proste przebiega inaczej niż dla samych wartości typów prostych.
- Pobieranie i ustawianie właściwości dla wartości typów prostych powoduje niejawnie tworzenie nakładek obiektowych.

## Sposób 5. Unikaj stosowania operatora == dla wartości o różnych typach

Jak myślisz, jaka będzie wartość poniższego wyrażenia?

```
"1.0e0" == { valueOf: function() { return true; } };
```

Te dwie pozornie niepowiązane wartości są uznawane przez operator == za równe, ponieważ — podobnie jak przy niejawnej konwersji typów opisanej w sposobie 3. — przed porównaniem obie są przekształcane na liczby. Łańcuch znaków "1.0e0" jest traktowany jak liczba 1, a obiekt zostaje przekształcony na liczbę w wyniku wywołania metody valueOf i przekształcenia jej wyniku (true) na liczbę, co także daje 1.

Kusząca jest myśl o wykorzystaniu tego rodzaju niejawnych konwersji w zadaniach takich jak odczyt wartości z pola formularza internetowego i porównywanie jej z liczbą:

```
var today = new Date();

if (form.month.value == (today.getMonth() + 1) &&
    form.day.value == today.getDate()) {
    // Wszystkiego najlepszego z okazji urodzin!
    // ...
}
```

Można jednak łatwo przekształcać wartości na liczby **jawnie** — za pomocą funkcji Number lub operatora jednoargumentowego +:

```
var today = new Date();

if (+form.month.value == (today.getMonth() + 1) &&
    +form.day.value == today.getDate()) {
    // Wszystkiego najlepszego z okazji urodzin!
    // ...
}
```

To bardziej przejrzyste rozwiązanie, ponieważ informuje czytelnika kodu o tym, jakie konwersje są przeprowadzane. Nie wymaga to zapamiętywania reguł konwersji. Jeszcze lepszym podejściem jest zastosowanie operatora **identyczności**:

```

var today = new Date();

if (+form.month.value === (today.getMonth() + 1) && // Sprawdzanie identyczności
    +form.day.value === today.getDate()) {          // Sprawdzanie identyczności
    // Wszystkiego najlepszego z okazji urodzin!
    // ...
}

```

Gdy argumenty są tego samego typu, operatory równości (==) i identyczności (===) działają tak samo. Dlatego jeśli wiesz, że argumenty mają ten sam typ, możesz zamiennie stosować te operatory. Jednak stosowanie operatora identyczności pozwala poinformować czytelników, że przy porównywaniu nie jest wykonywana konwersja. W przeciwnym razie czytelnik musi przypomnieć sobie reguły niejawnnej konwersji, aby zrozumieć, jak kod działa.

Okazuje się, że zasady niejawnnej konwersji nie są oczywiste. W tabeli 1.1 opisane są te reguły dla operatora == i argumentów różnych typów. Reguły są symetryczne; na przykład pierwsza z nich dotyczy zarówno porównania `null == undefined`, jak i porównania `undefined == null`. W trakcie konwersji zwykle następuje próba wygenerowania liczb. Jednak przy stosowaniu obiektów sytuacja się komplikuje. Następuje wtedy próba przekształcenia obiektu na wartość typu prostego w wyniku wywołania metody `valueOf` lub `toString`. Używana jest uzyskana w ten sposób wartość typu prostego. Dodatkową ciekawostką jest to, że dla obiektów typu `Date` te metody są stosowane w odwrotnej kolejności.

**Tabela 1.1.** Reguły niejawnnej konwersji stosowane przez operator ==

Typ pierwszego argumentu	Typ drugiego argumentu	Niejawna konwersja
<code>null</code>	<code>undefined</code>	Brak (wartość to zawsze <code>true</code> )
<code>null</code> lub <code>undefined</code>	Dowolny oprócz <code>null</code> i <code>undefined</code>	Brak (wartość to zawsze <code>false</code> )
Wartość typu prostego (łańcuch znaków, liczba lub wartość logiczna)	Obiekt typu <code>Date</code>	Typ prosty => liczba; obiekt typu <code>Date</code> => wartość typu prostego (sprawdzana metoda <code>toString</code> , a następnie <code>valueOf</code> )
Wartość typu prostego (łańcuch znaków, liczba lub wartość logiczna)	Obiekt inny niż <code>Date</code>	Typ prosty => liczba; obiekt typu innego niż <code>Date</code> => wartość typu prostego (sprawdzana metoda <code>valueOf</code> , a następnie <code>toString</code> )
Wartość typu prostego (łańcuch znaków, liczba lub wartość logiczna)	Wartość typu prostego (łańcuch znaków, liczba lub wartość logiczna)	Typ prosty => liczba

Operator `==` zwoźniczo pomija różnice w reprezentacji dat. Tego rodzaju rozwiązanie błędów można nazwać **semantyką typu „rób to, co mam na myśli”**. Jednak komputery tak naprawdę nie czytają w myślach. Istnieje zbyt wiele reprezentacji dat, aby JavaScript mógł określić, które z nich używasz. Możesz na przykład zakładać, że da się porównać łańcuch znaków z datą z obiektem typu `Date`:

```
var date = new Date("1999/12/31");
date == "1999/12/31"; // false
```

To porównanie zwróci wartość `false`, ponieważ obiekt typu `Date` po konwersji na tekst ma inny format niż łańcuch znaków z przykładu:

```
date.toString(); // "Fri Dec 31 1999 00:00:00 GMT-0800 (PST)"
```

Jednak ten błąd jest wynikiem bardziej ogólnego niezrozumienia niejawnej konwersji. Operator `==` nie potrafi określać ani ujednolicać dowolnych formatów dat. Przy jego stosowaniu niezbędne jest, aby zarówno autor, jak i czytelnicy kodu rozumieli skomplikowane zasady niejawnej konwersji. Lepszym podejściem jest jawna konwersja oparta na niestandardowym kodzie aplikacji i wykorzystanie operatora identyczności:

```
function toYMD(date) {
    var y = date.getFullYear() + 1900, // Lata są liczone od 1900
        m = date.getMonth() + 1,      // Miesiące są liczone od 0
        d = date.getDate();
    return y
        + "/" + (m < 10 ? "0" + m : m)
        + "/" + (d < 10 ? "0" + d : d);
}
toYMD(date) === "1999/12/31"; // true
```

Jawna konwersja gwarantuje, że nie pomylą Ci się reguły niejawnej konwersji, a także, co jeszcze lepsze, zwalnia czytelników z konieczności sprawdzania tych reguł lub ich zapamiętywania.

## Co warto zapamiętać?

- Z operatorem `==` związany jest skomplikowany zestaw niejawnych konwersji, przeprowadzanych, gdy argumenty są różnych typów.
- Stosuj operator `===`, aby jednoznacznie poinformować czytelników kodu, że w porównaniach nie są przeprowadzane niejawne konwersje.
- Przy porównywaniu wartości różnych typów stosuj jawną konwersję, aby działanie programu było bardziej zrozumiałe.

## Sposób 6. Ograniczenia mechanizmu automatycznego dodawania średników

Jednym z udogodnień z JavaScriptu jest możliwość pomijania średników kończących instrukcje. Dzięki temu kod staje się bardziej estetyczny:

```
function Point(x, y) {  
    this.x = x || 0  
    this.y = y || 0  
}  
  
Point.prototype.isOrigin = function() {  
    return this.x === 0 && this.y === 0  
}
```

Ten zapis jest możliwy dzięki **automatycznemu dodawaniu średników**. Jest to technika parsowania programów, która w określonych kontekstach potrafi wykryć brakujące średniki i automatycznie dodać je do programu. W standardzie ECMAScript ten mechanizm jest dokładnie opisany, dlatego średniki można pomijać w różnych silnikach JavaScriptu.

Jednak z dodawaniem średników (podobnie jak z niejawnymi konwersjami opisanymi w sposobach 3. i 5.) związane są pewne pułapki, dlatego konieczne należy poznać reguły działania tego mechanizmu. Nawet jeśli nigdy nie pomijasz średników, w składni JavaScriptu występują pewne ograniczenia wynikające z omawianego mechanizmu. Dobra wiadomość jest taka, że gdy poznasz zasady dodawania średników, będziesz mógł swobodnie pomijać ich zbędne wystąpienia.

Oto pierwsza reguła dodawania średników:

*Średniki są dodawane tylko przed symbolem }, po znaku nowego wierszu (lub kilku takich znakach) i na końcu kodu programu.*

Oznacza to, że możesz pomijać średniki tylko na końcu wiersza, bloku lub programu. Poniższe funkcje są więc poprawne:

```
function square(x) {  
    var n = +x  
    return n * n  
}  
function area(r) { r = +r; return Math.PI * r * r }  
function add1(x) { return x + 1 }
```

Jednak poniższy kod jest nieprawidłowy:

```
function area(r) { r = +r return Math.PI * r * r } // Błąd
```

Oto druga reguła dodawania średników:

*Średniki są dodawane tylko wtedy, gdy w wyniku parsowania następny symbol jest uznawany za niedozwolony.*

Tak więc dodawanie średników to mechanizm **naprawiania błędów**. Oto prosty przykładowy fragment kodu:

```
a = b
(f());
```

W wyniku parsowania jest on traktowany jako jedna poprawna instrukcja:

```
a = b(f());
```

Średnik nie jest wtedy wstawiany. Przyjrzyj się teraz innemu fragmentowi:

```
a = b
f();
```

Ten kod jest parsowany jako dwie odrębne instrukcje, ponieważ polecenie:

```
a = b f();
```

powoduje błąd parsowania.

Z tej reguły wynikają niefortunne skutki — zawsze musisz zwracać uwagę na początek następnej instrukcji, aby wykryć, czy możesz bezpiecznie pominąć średnik. Nie możesz opuścić średnika, jeśli początkowy symbol z następnego wiersza można uznać za kontynuację instrukcji.

Jest pięć problematycznych znaków, na które trzeba uważać: (, [, +, - i /. Każdy z nich w zależności od kontekstu może być zarówno operatorem w wyrażeniu, jak i przedrostkiem instrukcji. Zwracaj więc uwagę na polecenia kończące się wyrażeniem (takie jak przypisanie we wcześniejszym przykładzie). Jeśli następny wiersz kończy się jednym z tych pięciu problematycznych znaków, średnik nie zostanie wstawiony. Najczęściej zdarza się to wtedy, gdy instrukcja rozpoczyna się nawiasem, tak jak w przedstawionym przykładzie. Innym często występującym scenariuszem jest używanie literału tablicowego:

```
a = b
["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

Wygląda to jak dwie instrukcje — przypisanie, po którym następuje wywołanie funkcji po kolei dla łańcuchów znaków "r", "g" i "b". Jednak ponieważ druga instrukcja rozpoczyna się od znaku [, w wyniku parsowania powstaje jedno polecenie:

```
a = b["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

Jeśli wyrażenie w nawiasie kwadratowym wydaje Ci się dziwne, pamiętaj, że JavaScript zezwala na tworzenie wyrażeń rozdzielonych przecinkami. Są one przetwarzane od lewej do prawej i zwracają wartość ostatniego podwyrażenia, którą tu jest łańcuch znaków "b".

Symbole `+`, `-` i `/` rzadziej występują na początku instrukcji, jednak czasem to się zdarza. Ciekawy w tym kontekście jest zwłaszcza symbol `/`. Na początku instrukcji nie jest on całym symbolem, a tylko początkiem wyrażenia regularnego:

```
/Error/i.test(str) && fail();
```

Ta instrukcja sprawdza, czy łańcuch znaków pasuje do wyrażenia nieregularnego `/Error/i` (wielkość znaków nie ma tu znaczenia). Jeśli znajdzie pasujący tekst, wywoływana jest funkcja `fail`. Założmy teraz, że ten kod znajduje się po przypisaniu niezakończonym średnikiem:

```
a = b  
/Error/i.test(str) && fail();
```

Wtedy kod w wyniku parsowania jest uznawany za jedną instrukcję:

```
a = b / Error / i.test(str) && fail();
```

Oznacza to, że początkowy symbol `/` jest traktowany jako operator dzielenia!

Doświadczeni programiści JavaScriptu uczą się patrzeć na wiersz pod daną instrukcją, gdy chcą pominąć średnik. W ten sposób upewniają się, że w wyniku parsowania instrukcja nie stanie się błędna. Należy o tym pamiętać także w trakcie refaktoryzacji. Oto przykładowy w pełni poprawny program, w którym dodawane są trzy średniki:

```
a = b // Dodawany średnik  
var x // Dodawany średnik  
(f()) // Dodawany średnik
```

Może się on nieoczekiwanie zmienić w inny program, w którym dodawane są tylko dwa średniki:

```
var x // Dodawany średnik  
a = b // Średnik nie jest dodawany  
(f()) // Dodawany średnik
```

Choć przeniesienie instrukcji `var` o jeden wiersz nie powinno niczego zmieniać (szczegółowy opis zasięgu zmiennych zawiera Sposób 12.), występowanie nawiasów po `b` sprawia, że program jest błędnie parsowany do następującej postaci:

```
var x;  
a = b(f());
```

Dlatego zawsze musisz pamiętać o pominiętych średnikach i sprawdzać, czy na początku następnego wiersza nie znajdują się symbole, które blokują dodawanie średników. Inną możliwość to poprzedzanie instrukcji rozpoczynających się od symboli `(`, `[`, `+`, `-` i `/` średnikiem. Na przykład wcześniej przedstawiony przykład można zmodyfikować, aby zabezpieczyć wywołanie funkcji umieszczone w nawiasie:

```
a = b    // Dodawany średnik
var x    // Średnik w następnym wierszu
:(f())   // Dodawany średnik
```

Teraz możesz bezpiecznie przenieść deklarację `var` na początek kodu bez obaw o to, że zmienisz działanie programu:

```
var x    // Dodawany średnik
a = b    // Średnik w następnym wierszu
:(f())   // Dodawany średnik
```

Inna sytuacja, w której pominięcie średnika może prowadzić do problemów, związana jest ze scalaniem skryptów (zobacz Sposób 1.). Każdy plik może obejmować wiele wyrażeń IIFE (więcej informacji na temat takich wyrażeń znajdziesz w sposobie 13.):

```
// file1.js
(function() {
    // ...
})();
```

```
// file2.js
(function() {
    // ...
})();
```

Gdy każdy plik jest wczytywany jako odrębny program, na końcu automatycznie wstawiany jest średnik, co powoduje przekształcenie wywołania funkcji w instrukcję. Inaczej jest jednak przy złączaniu plików:

```
(function() {
    // ...
})();
(function() {
    // ...
})();
```

W efekcie wszystkie wywołania są traktowane jak jedna instrukcja:

```
(function() {
    // ...
})();(function() {
    // ...
})();
```

Dlatego gdy pomijasz średnik w instrukcji, musisz sprawdzić nie tylko następny symbol w bieżącym pliku, ale też wszystkie symbole, które *mogą* znaleźć się po danej instrukcji w wyniku scalenia skryptów. Można wtedy zastosować technikę podobną do opisanej wcześniej i chronić skrypty przed nieostrożnym scalaniem dzięki zapobiegawczemu poprzedzaniu każdego pliku dodatkowym średnikiem — przynajmniej w sytuacji, gdy pierwsza instrukcja rozpoczyna się jednym z niebezpiecznych znaków (`(`, `[`, `+`, `-` lub `/`):



```
// file1.js
:(function() {
    // ...
})();

// file2.js
:(function() {
    // ...
})();
```

To sprawia, że nawet jeśli w poprzedzającym pliku brakuje końcowego średnika, to po połączeniu skryptów instrukcje nadal będą traktowane jako odrębne polecenia:

```
:(function() {
    // ...
})();
:(function() {
    // ...
})();
```

Oczywiście lepiej jest, gdy w procesie złączania skryptów automatycznie dodawane są średniki między plikami. Jednak nie wszystkie narzędzia do złączania skryptów są odpowiednio napisane, dlatego najbezpieczniej jest zapobiegawczo dodać średniki.

Możliwe, że zastanawiasz się teraz, czy to nie za dużo kłopotów. W końcu jeśli nigdy nie będziesz pomijał średników, nie narazisz się na problemy, prawda? Nie do końca. Czasem JavaScript wstawia średnik nawet wtedy, gdy wydaje się, że błąd parsowania nie występuje. Dotyczy to **ograniczonych konstrukcji** (ang. *restricted productions*) w składni JavaScriptu, w których między dwoma symbolami nie mogą występować znaki nowego wiersza. Najbardziej ryzykowna jest tu instrukcja `return`; między słowem kluczowym `return` a opcjonalnymi argumentami nie może znajdować się znak nowego wiersza. Dlatego poniższe polecenie zwraca nowy obiekt:

```
return { };
```

Inaczej jest z następnym fragmentem kodu:

```
return
{ };
```

Ten kod w wyniku parsowania jest traktowany jak trzy odrębne polecenia:

```
return;
{ }
;
```

Oznacza to, że znak nowego wiersza po słowie kluczowym `return` wymusza automatyczne dodanie średnika. W wyniku parsowania powstaje instrukcja `return` bez argumentu, pusty blok i pusta instrukcja. Oto inne ograniczone konstrukcje:

- instrukcja `throw`,
- instrukcja `break` lub `continue` z jawnie podaną etykietą,
- operator przyrostkowy `++` lub `--`.

Ta ostatnia reguła umożliwia jednoznaczną interpretację fragmentów kodu takich jak poniższy:

```
a
++
b
```

Operator `++` może być zarówno przedrostkiem, jak i przyrostkiem, jednak w tym drugim przypadku nie może być poprzedzony znakiem nowego wiersza. Dlatego ten kod po parsowaniu przyjmuje następującą postać:

```
a; ++b;
```

Trzecia i ostatnia reguła dodawania średników brzmi tak:

*Średniki nigdy nie są dodawane jako separatory w nagłówkach pętli `for` lub pustych instrukcjach.*

To oznacza, że zawsze trzeba jawnie dodawać średniki w nagłówku pętli `for`. Jeśli tego nie zrobisz, kod podobny do poniższego spowoduje błąd parsowania:

```
for (var i = 0, total = 1 // Błąd parsowania
    i < n
    i++) {
    total *= i
}
```

Podobnie w pętli z pustym ciałem niezbędny jest jawnie podany średnik. W przeciwnym razie wystąpi błąd parsowania:

```
function infiniteLoop() { while(true) } // Błąd parsowania
```

Jest to jedna z sytuacji, w których średnik jest konieczny:

```
function infiniteLoop() { while(true); }
```

## Co warto zapamiętać?

- Średniki są dodawane tylko przed symbolem `}`, na końcu wiersza lub na końcu programu.
- Średniki są dodawane tylko wtedy, gdy w wyniku parsowania stwierdzono, że następny symbol jest niedozwolony.
- Nigdy nie pomijaj średnika przed instrukcjami rozpoczynającymi się od symboli `(`, `[`, `+`, `-` lub `/`.
- Gdy scalasz skrypty, wstawiaj jawnie średniki na początku skryptów.

- Nigdy nie dodawaj nowego wiersza przed argumentami instrukcji `return`, `throw`, `break`, `continue`, `++` lub `--`.
- Średniki nigdy nie są dodawane jako separatory w nagłówkach pętli `for` lub w pustych instrukcjach.

## Sposób 7. Traktuj łańcuchy znaków jak sekwencje 16-bitowych jednostek kodowych

Zestaw znaków Unicode jest uważany za skomplikowany. Mimo powszechnego stosowania łańcuchów znaków większość programistów nie uczy się tego standardu i liczy na to, że kod będzie działał poprawnie. Jednak nie ma się czego obawiać. Podstawy działania Unicode są bardzo proste — wszystkim jednostkom tekstu z każdego systemu pisma z całego świata przypisana jest liczba całkowita z przedziału od 0 do 1 114 111. W terminologii związanej z Unicode ta liczba to **współrzędna kodowa znaku** (ang. *code point*). I to wszystko. Nie różni się to od innych zestawów znaków, takich jak ASCII. Różnica polega na tym, że w ASCII każdemu indeksowi odpowiada niepowtarzalna reprezentacja binarna. W Unicode do współrzędnych kodowych mogą być przypisane różne reprezentacje binarne. W poszczególnych kodowaniach zdecydowano się na różne kompromisy związane z ilością pamięci potrzebnej na łańcuchy znaków i szybkością wykonywania operacji (takich jak obsługa indeksów łańcuchów znaków). Obecnie istnieje wiele standardowych kodowań zestawu znaków Unicode. Najpopularniejsze z nich to UTF-8, UTF-16 i UTF-32.


Dodatkowym źródłem komplikacji jest to, że projektanci zestawu Unicode początkowo nie doszacowali liczby potrzebnych współrzędnych kodowych. Pierwotnie uważano, że w zestawie Unicode wystarczy mniej niż  $2^{16}$  takich współrzędnych. To sprawiło, że UCS-2, początkowe standardowe kodowanie 16-bitowe, było wyjątkowo atrakcyjnym rozwiązaniem. Ponieważ każda współrzędna bitowa mieściła się w 16-bitowej liczbie, występowało proste odwzorowanie jeden do jednego między współrzędnymi kodowymi a elementami kodowania, czyli **jednostkami kodowymi** (ang. *code units*). Kodowanie UCS-2 obejmowało odrębne 16-bitowe jednostki kodowe, z których każda odpowiadała jednej współrzędnej kodowej z zestawu Unicode. Główną zaletą tego kodowania jest to, że obsługa indeksów w łańcuchach znaków to mało kosztowna operacja wykonywana w stałym czasie. Dostęp do  $n$ -tej współrzędnej kodowej wymaga tylko wybrania  $n$ -tego 16-bitowego elementu tablicy. Rysunek 1.1 pokazuje przykładowy łańcuch znaków zawierający tylko współrzędne kodowe z pierwotnego 16-bitowego zakresu. Jak widać, w tym łańcuchu znaków Unicode indeksy jednostek kodowych pasują do współrzędnych kodowych.


'h'	'e'	'l'	'l'	'o'
0x0068	0x0065	0x006c	0x006c	0x006f
0	1	2	3	4

**Rysunek 1.1.** Łańcuch znaków w języku JavaScript zawierający współrzędne kodowe z kodowania BMP

Dlatego początkowo w wielu platformach stosowano dla łańcuchów znaków kodowanie 16-bitowe. Jedną z tych platform była Java. W JavaScriptcie zastosowano to samo podejście. Każdy element w łańcuchach znaków z JavaScriptu to wartość 16-bitowa. Gdyby Unicode zachował postać z początku lat 90., każdy element w łańcuchach JavaScriptu wciąż odpowiadałby jednej współrzędnej kodowej.

Ten 16-bitowy zakres jest dość pojemny i obejmuje znacznie więcej systemów pisma niż ASCII lub jakiekolwiek z późniejszych odmian tego zestawu. Jednak później stało się jasne, że wymagania zestawu Unicode wykraczają poza ten początkowy zakres. Ten zestaw został rozwinięty i obecnie obejmuje ponad  $2^{20}$  współrzędnych kodowych. Nowy, powiększony zakres jest podzielony na 17 podzakresów po  $2^{16}$  współrzędnych kodowych każdy. Pierwszy z tych podzakresów, BMP (ang. *Basic Multilingual Plane*), obejmuje pierwotne  $2^{16}$  współrzędnych kodowych. Pozostałych 16 podzakresów to *przestrzenie dodatkowe* (ang. *supplementary planes*).

Po rozszerzeniu zakresu współrzędnych kodowych kodowanie UCS-2 stało się przestarzałe. Trzeba je było powiększyć, aby reprezentowało dodatkowe współrzędne kodowe. Następca UCS-2, kodowanie UTF-16, działa bardzo podobnie, ale dodatkowo obejmuje *pary surogatów*. Są to pary 16-bitowych jednostek kodowych, które wspólnie określają jedną współrzędną kodową z zakresu od  $2^{16}$  w górę. Na przykład symbol klucza G („”) jest powiązany ze współrzędną kodową U+1D11E (to szesnastkowy zapis współrzędnej kodowej 119 070). W kodowaniu UTF-16 ten symbol jest reprezentowany za pomocą pary jednostek kodowych 0xd834 i 0xdd1e. Tę współrzędną kodową można odkodować dzięki połączeniu wybranych bitów z obu jednostek kodowych. Pomysłowe jest to, że to kodowanie gwarantuje, iż żaden z surogatów nie zostanie pomyłony z poprawną współrzędną kodową z kodowania BMP. Dzięki temu zawsze wiadomo, czy używany jest surogat — nawet jeśli wyszukiwanie rozpoczyna się w środkowej części łańcucha znaków. Przykładowy łańcuch znaków z parą surogatów jest przedstawiony na rysunku 1.2. Pierwsza współrzędna kodowa z tego łańcucha znaków wymaga pary surogatów, dlatego indeksy jednostek kodowych są inne niż indeksy współrzędnych kodowych.

'  '	' '	'c'	'l'	'e'	'f'	
0xd834	0xdd1e	0x0020	0x0063	0x006c	0x0065	0x0066
0	1	2	3	4	5	6


**Rysunek 1.2.** Łańcuch znaków w języku JavaScript obejmujący współrzedną kodową z przestrzeni dodatkowej

Ponieważ każda współrzedna kodowa w kodowaniu UTF-16 może wymagać jednej lub dwóch 16-bitowych jednostek kodowych, UTF-16 to kodowanie o **zmiennej długości**. Miejsce zajmowane w pamięci przez łańcuch znaków o długości  $n$  zależy od używanych współrzednych kodowych. Ponadto znalezienie  $n$ -tej współrzednej kodowej łańcucha znaków nie jest już operacją wykonywaną w stałym czasie. Zwykle wymaga to szukania jej od początku łańcucha.





Jednak do czasu rozszerzenia zestawu Unicode w JavaScriptcie wprowadzono już 16-bitowe elementy łańcuchów znaków. Właściwości i metody łańcuchów znaków, na przykład `length`, `charAt` i `charCodeAt`, działają na poziomie *jednostek* kodowych, a nie *współrzednych* kodowych. Dlatego gdy łańcuch znaków zawiera współrzedne kodowe z przestrzeni dodatkowych, JavaScript reprezentuje każdą taką współrzedną za pomocą dwóch elementów — pary surogatów z kodowania UTF-16 odpowiadającej danej współrzednej kodowej. Można ująć to w prosty sposób:

*Element łańcuch znaków w JavaScriptcie to 16-bitowa jednostka kodowa.*

Wewnętrznie silniki JavaScriptu mogą optymalizować przechowywanie wartości łańcuchów znaków. Jednak dla właściwości i metod łańcuchy znaków to sekwencje jednostek kodowych UTF-16. Wróć do łańcucha znaków z rysunku 1.2. Choć zawiera on sześć współrzednych kodowych, JavaScript informuje, że długość tego łańcucha to siedem znaków:

```
 clef".length; // 7
"G clef".length; // 6
```

Gdy pobierasz poszczególne elementy łańcucha znaków, zwracane są jednostki kodowe, a nie współrzedne kodowe:

```
 clef".charCodeAt(0); // 55348 (0xd834)
 clef".charCodeAt(1); // 56606 (0xdd1e)
 clef".charAt(1) === " "; // false
 clef".charAt(2) === " "; // true
```

Także wyrażenia regularne działają na poziomie jednostek kodowych. Wzorec reprezentujący jeden znak (.) pasuje do pojedynczej jednostki kodowej:

```
/^.$/.test("a"); // false  
/^.$/.test("aa"); // true
```

To rozwiązanie oznacza, że aplikacje korzystające z pełnego zakresu znaków Unicode muszą wykonywać dużo więcej pracy. Nie mogą polegać na metodach łańcuchów znaków, długościach, wyszukiwaniu opartym na indeksach i wielu wyrażeniach regularnych. Jeśli używasz znaków spoza zakresu BMP, warto poszukać bibliotek obsługujących współrzędne kodowe. Trudno jest poprawnie zarządzać szczegółami kodowania i dekodowania, dlatego warto wykorzystać istniejące biblioteki, zamiast samodzielnie pisać potrzebny kod.

Choć wbudowane łańcuchy znaków z JavaScriptu działają na poziomie jednostek kodowych, interfejsy API potrafią obsługiwać współrzędne kodowe i pary surogatów. Niektóre biblioteki ze standardu ECMAScript poprawnie obsługują pary surogatów. Potrafią to na przykład funkcje do manipulowania identyfikatorami URI (`encodeURIComponent`, `decodeURIComponent` i `decodeURIComponent`). Jeśli w JavaScriptcie dostępna jest biblioteka działająca na łańcuchach znaków, przeznaczona na przykład do manipulowania zawartością stron internetowych lub wykonywania operacji wejścia-wyjścia, powinieneś zapoznać się z jej dokumentacją i sprawdzić, jak obsługiwane są współrzędne kodowe z pełnego zestawu Unicode.

### Co warto zapamiętać?

- Łańcuchy znaków w JavaScriptcie składają się z 16-bitowych jednostek kodowych, a nie ze współrzędnych kodowych Unicode.
- Współrzędne kodowe Unicode z zakresu od  $2^{16}$  wzwyż są reprezentowane w JavaScriptcie za pomocą dwóch jednostek kodowych (pary surogatów).
- Pary surogatów powodują błędy w zliczaniu elementów łańcuchów znaków. Wpływa to na pracę metod `length`, `charAt` i `charCodeAt`, a także na wzorce wyrażen regularnych (na przykład „.”).
- Jeśli chcesz manipulować łańcuchami znaków z uwzględnieniem współrzędnych kodowych, stosuj niezależne biblioteki.
- Gdy używasz biblioteki manipulującej łańcuchami znaków, zapoznaj się z dokumentacją, aby sprawdzić, jak obsługiwane są współrzędne kodowe z pełnego zakresu Unicode.

# 2

## Zasięg zmiennych

Zasięg jest dla programistów jak tlen — znajduje się wszędzie i zazwyczaj nawet się o nim nie myśli. Jeśli jednak go zabraknie, zaczniesz się dusić.

Dobra wiadomość jest taka, że w JavaScriptcie podstawowe zasady określania zasięgu są proste, dobrze zaprojektowane i dające bardzo duże możliwości. Istnieją jednak wyjątki. Aby efektywnie posługiwać się JavaScriptem, musisz opanować podstawowe zagadnienia związane z zasięgiem zmiennych, a także nietypowe przypadki, które mogą prowadzić do subtelnych, ale poważnych problemów.

### Sposób 8. Minimalizuj liczbę obiektów globalnych

W JavaScriptcie można łatwo tworzyć zmienne w globalnej przestrzeni nazw. Tworzenie zmiennych globalnych jest proste, ponieważ nie wymagają one deklaracji i są automatycznie dostępne w całym kodzie programu. Ta wygoda sprawia, że stosowanie zmiennych globalnych jest kuszące dla początkujących. Jednak doświadczeni programiści wiedzą, że należy unikać takich zmiennych. Zmienne globalne zaśmiecają wspólną przestrzeń nazw, z której korzystają wszyscy autorzy danego programu, i mogą prowadzić do przypadkowych kolizji nazw. Zmienne globalne są niezgodne z modułowością. Powodują niepotrzebne wiązania między odrębnymi komponentami programów. Choć podejście „napisz teraz, a później uporządkuj” jest wygodne, najlepsi programiści nieustannie zwracają uwagę na strukturę swoich programów i grupują związane funkcje oraz rozdzielają niezwiązane komponenty w ramach procesu programowania.

Ponieważ globalna przestrzeń nazw to jedyny praktyczny mechanizm komunikowania się odrębnych komponentów programów w JavaScriptcie, czasem trzeba z niej korzystać. W komponentach lub bibliotekach trzeba definiować globalne nazwy, aby inne części programów mogły z nich korzystać. Jednak

w innych sytuacjach najlepiej jest dbać o to, żeby zmienne były jak najbardziej lokalne. Oczywiście *możliwe* jest napisanie programu z samymi zmiennymi globalnymi, jest to jednak prosta droga do problemów. Nawet w bardzo prostych funkcjach z definicjami ich zmiennych tymczasowych trzeba wtedy sprawdzać, czy w innym kodzie nie zastosowano identycznych nazw:

```
var i, n, sum; // Zmienne globalne
function averageScore(players) {
    sum = 0;
    for (i = 0, n = players.length; i < n; i++) {
        sum += score(players[i]);
    }
    return sum / n;
}
```

Ta definicja funkcji `averageScore` spowoduje problemy, jeśli w użytej tu funkcji `score` wykorzystywane są zmienne globalne o identycznych nazwach:

```
var i, n, sum; // Te same zmienne globalne co w funkcji averageScore!
function score(player) {
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

Rozwiązaniem jest tworzenie zmiennych lokalnych w kodzie, który ich potrzebuje:

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

function score(player) {
    var i, n, sum;
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

Globalna przestrzeń nazw JavaScriptu jest też udostępniana jako **obiekt globalny**, dostępny na początku programu jako pierwotna wartość słowa kluczowego `this`. W przeglądarce internetowej ten globalny obiekt jest związany ze zmienną globalną `window`. Dodanie lub zmodyfikowanie zmiennych globalnych oznacza automatyczną aktualizację obiektu globalnego:



```
this.foo; // undefined  
foo = "global foo";  
this.foo; // "global foo"
```

Podobnie aktualizacja obiektu globalnego oznacza automatyczną aktualizację globalnej przestrzeni nazw:

```
var foo = "global foo";  
this.foo = "zmodyfikowano";  
foo; // "zmodyfikowano"
```

Istnieją więc dwa mechanizmy tworzenia zmiennych globalnych: możesz deklarować je za pomocą słowa kluczowego `var` w zasięgu globalnym lub dodawać do obiektu globalnego. Obie techniki działają poprawnie, jednak użycie słowa kluczowego `var` ma tę zaletę, że jednoznacznie określony jest wpływ instrukcji na zasięg programu. Próba użycia niezwiązanej zmiennej prowadzi do błędu czasu wykonania, a jednoznaczne i proste zarządzanie zasięgiem ułatwia czytelnikom kodu zrozumienie, jakie zmienne globalne są zadeklarowane.

Choć najlepiej jest ograniczyć korzystanie z obiektu globalnego, istnieje zastosowanie, w którym jest on niezastąpiony. Ponieważ obiekt globalny pozwala dynamicznie sprawdzać środowisko globalne, można wykorzystać go do badania środowiska wykonawczego i wykrywania, jakie funkcje są dostępne w danej platformie. Na przykład w standardzie ES5 wprowadzono nowy obiekt globalny `JSON`, który służy do wczytywania i zapisywania danych w formacie JSON. Aby uniknąć wykonywania kodu w środowisku, w którym ten obiekt jest niedostępny, możesz wykrywać jego dostępność i zapewniać alternatywną implementację w sytuacji, gdy nie można użyć tego obiektu:

```
if (!this.JSON) {  
  this.JSON = {  
    parse: ...,  
    stringify: ...  
  };  
}
```

Jeśli udostępniasz implementację obiektu `JSON`, oczywiście możesz zawsze z niej korzystać. Jednak prawie zawsze lepiej jest używać implementacji wbudowanych w środowisko hosta, ponieważ są dobrze przetestowane pod kątem poprawności i zgodności ze standardami, a także często zapewniają wyższą wydajność niż implementacje niezależnych producentów.

Pokazana tu technika wykrywania dostępności funkcji jest ważna zwłaszcza w przeglądarkach internetowych, ponieważ ten sam kod może być wykonywany przez bardzo różne przeglądarki i ich wersje. Wykrywanie dostępności funkcji to stosunkowo łatwa metoda zapewniania odporności programów na różnice w mechanizmach oferowanych przez różne platformy. Ta technika jest przydatna także w innych sytuacjach, na przykład w bibliotekach, które mogą działać zarówno w przeglądarce, jak i w środowiskach serwerowych wykorzystujących JavaScript.

## Co warto zapamiętać?

- Unikaj deklarowania zmiennych globalnych.
- Deklaruj zmienne na jak najbardziej lokalnym poziomie.
- Unikaj dodawania właściwości do obiektu globalnego.
- Używaj obiektu globalnego do wykrywania funkcji dostępnych w platformach.

## Sposób 9. Zawsze deklaruj zmienne lokalne

Jeśli istnieje coś powodującego więcej problemów niż zmienna globalna, to *niecelowo* utworzona zmienna globalna. Niestety, obowiązujące w JavaScriptcie zasady przypisywania sprawiają, że łatwo można przypadkowo utworzyć zmienne globalne. Zamiast zgłaszać błąd, program przypisujący wartość do niezwiązanej zmiennej tworzy nową zmienną globalną i ustawia jej wartość. To oznacza, że jeśli zapomnisz zadeklarować zmienną lokalną, program niezauważalnie przekształci ją w zmienną globalną:

```
function swap(a, i, j) {  
    temp = a[i]; // Zmienna globalna  
    a[i] = a[j];  
    a[j] = temp;  
}
```

Ten program zostanie wykonany bez zgłaszania błędów, choć brak deklaracji `var` przy zmiennej `temp` prowadzi do przypadkowego utworzenia zmiennej globalnej. We właściwej implementacji zmienna `temp` jest zadeklarowana za pomocą słowa kluczowego `var`:

```
function swap(a, i, j) {  
    var temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```

Celowe tworzenie zmiennych globalnych to oznaka złego stylu, natomiast przypadkowe dodawanie takich zmiennych może doprowadzić do prawdziwej katastrofy. Dlatego wielu programistów używa narzędzi typu *lint*, które analizują kod źródłowy pod kątem złego stylu lub potencjalnych błędów i często potrafią wskazywać miejsca użycia niezwiązanych zmiennych. Narzędzie typu *lint* wykrywające niezadeklarowane zmienne pobiera podany przez użytkownika zestaw znanych zmiennych globalnych (występujących w środowisku hosta lub zdefiniowanych w odrębnych plikach), a następnie informuje o referencjach lub przypisaniach zmiennych, które ani nie znajdują się na podanej liście, ani nie są zadeklarowane w programie. Warto poświęcić trochę czasu na zbadanie, jakie narzędzia programistyczne związane z JavaScriptem są

dostępne. Wbudowanie w proces programowania automatycznego wykrywania często spotykanych błędów, na przykład przypadkowego tworzenia zmiennych lokalnych, pomoże Ci uniknąć wielu problemów.

### Co warto zapamiętać?

- Zawsze deklaruj nowe zmienne lokalne za pomocą słowa kluczowego `var`.
- Pomyśl o wykorzystaniu narzędzi typu `lint` do wykrywania niezwiązanych zmiennych.

## Sposób 10. Unikaj słowa kluczowego with

Biedne `with` — prawdopodobnie żaden inny mechanizm JavaScriptu nie ma tak złej opinii. Jest to jednak uzasadnione. Wszelkie udogodnienia, jakie to słowo kluczowe zapewnia, błędą w obliczu jego zawodności i niewydajności.

Przyczyny stosowania `with` są zrozumiałe. Programy często muszą wywoływać po kolei różne metody tego samego obiektu. Unikanie wielokrotnych referencji do obiektu jest wygodne:

```
function status(info) {  
  var widget = new Widget();  
  with (widget) {  
    setBackground("blue");  
    setForeground("white");  
    setText("Status: " + info); // Niejednoznaczna referencja  
    show();  
  }  
}
```

Kuszący jest też pomysł zastosowania `with` do importowania zmiennych z obiektów używanych jako moduły:

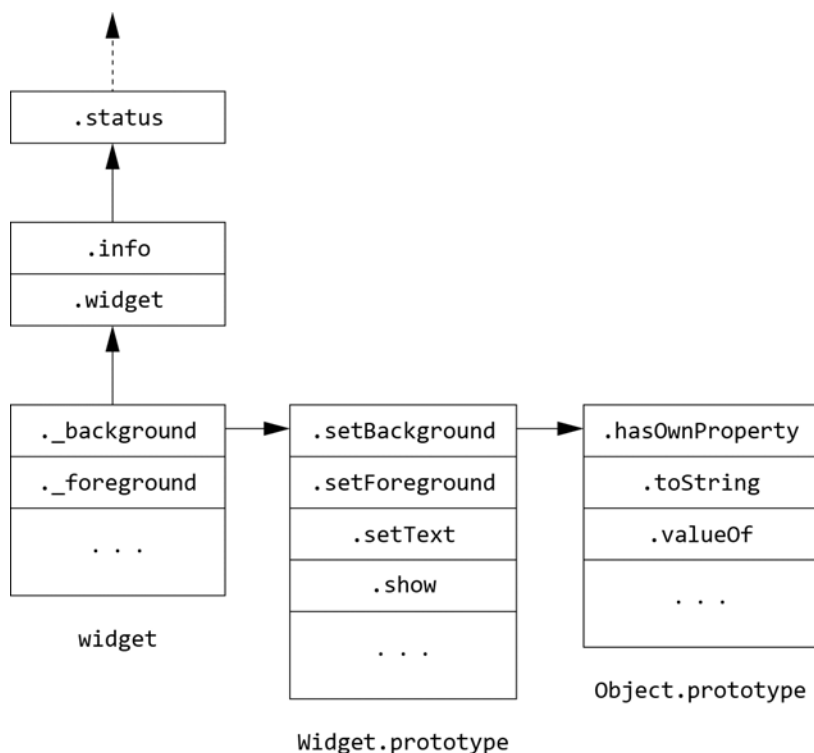
```
function f(x, y) {  
  with (Math) {  
    return min(round(x), sqrt(y)); // Niejednoznaczne referencje  
  }  
}
```

W obu sytuacjach słowo kluczowe `with` umożliwia łatwe pobieranie właściwości obiektu i ustawianie ich jako zmiennych lokalnych w bloku.

Przedstawione przykłady wyglądają zachęcająco. Jednak żaden z nich nie działa zgodnie z oczekiwaniami. Zauważ, że w obu przykładach używane są dwa różne rodzaje zmiennych — wskazujące właściwości obiektu podanego po słowie kluczowym `with` (`setBackground`, `round` i `sqrt`) i wskazujące zewnętrznie związane zmienne (`info`, `x` i `y`). Jednak w składni te elementy niczym się nie różnią — i jedno, i drugie wyglądają jak zmienne.

JavaScript traktuje wszystkie zmienne identycznie. Wyszukuje je w zasięgu, zaczynając od najbardziej lokalnego i przechodząc na zewnątrz. Instrukcja `with` traktuje obiekt tak, jakby reprezentował zasięg zmiennych, dlatego w bloku tej instrukcji wyszukiwanie najpierw dotyczy właściwości o podanej nazwie. Jeśli w danym obiekcie nie istnieje właściwość o tej nazwie, wyszukiwanie jest kontynuowane w zasięgu zewnętrznym.

Rysunek 2.1 przedstawia diagram wewnętrznej reprezentacji (z silnika JavaScriptu) zasięgu funkcji `status` w czasie wykonywania instrukcji `with`. W specyfikacji standardu ES5 jest to **środowisko leksykalne** (ang. *lexical environment*; w starszych wersjach standardu używane jest określenie **łańcuch zasięgu** — ang. *scope chain*). Wewnętrzny zasięg środowiska jest określany przez obiekt `widget`. Następny zasięg obejmuje wiązania zmiennych lokalnych `info` i `widget` funkcji. Na następnym poziomie znajduje się wiązanie funkcji `status`. Zauważ, że w normalnym zasięgu na tym poziomie środowiska występuje tyle wiązań, ile jest zmiennych w zasięgu lokalnym. Jednak w zasięgu określonym dla instrukcji `with` zbiór wiązań zależy od tego, co znajduje się w obiekcie w danym momencie.



**Rysunek 2.1.** Środowisko leksykalne (łańcuch zasięgu) funkcji `status`

Jaką masz pewność, że określone właściwości znajdują się (lub nie) w obiekcie podanym w instrukcji `with`? W bloku `with` każda referencja do zmiennej zewnętrznej wymaga przyjęcia założenia, że właściwość o tej samej nazwie nie występuje w podanym obiekcie ani *w żadnym z jego prototypów*. Inne części programu, które tworzą lub modyfikują obiekt podany w instrukcji `with` lub jego prototypy, mogą działać niezgodnie z tym założeniem. Niepożądana jest sytuacja, gdy ktoś musi sprawdzać lokalny kod, aby ustalić, jakie zmienne lokalne są w nim używane.

Możliwe konflikty między przestrzeniami nazw z zasięgu zmiennych i obiektu sprawiają, że bloki `with` są bardzo narażone na problemy. Na przykład: jeśli do obiektu `widget` z przedstawionego przykładu dodana zostanie właściwość `info`, to w funkcji `status` użyta zostanie ta właściwość zamiast parametru `info` funkcji. Może się to zdarzyć w wyniku zmodyfikowania kodu źródłowego, jeśli programista zdecyduje, że wszystkie obiekty `widget` powinny mieć właściwość `info`. Co gorsza, ktoś może dodać właściwość `info` do prototypowego obiektu `Widget` w czasie wykonywania programu, co spowoduje, że funkcja `status` zacznie sprawiać problemy w nieoczekiwanych miejscach:

```
status("connecting"); // Status: connecting
Widget.prototype.info = "[[widget info]]";
status("connected"); // Status: [[widget info]]
```

Podobnie funkcja `f` może przestać działać, jeśli ktoś doda do obiektu `Math` właściwość `x` lub `y`:

```
Math.x = 0;
Math.y = 0;
f(2, 9); // 0
```

Jest mało prawdopodobne, że ktoś doda właściwości `x` i `y` do obiektu `Math`. Jednak nie zawsze można łatwo przewidzieć, czy dany obiekt zostanie zmodyfikowany lub czy będzie miał nieznane Ci właściwości. Ponadto okazuje się, że mechanizm nieprzewidywalny dla ludzi może być taki także dla optymalizatorów. Zasięgi w JavaScriptcie zwykle można przedstawić za pomocą wydajnych wewnętrznych struktur danych, co pozwala na szybkie wyszukiwanie zmiennych. Jednak ponieważ blok `with` wymaga przeszukiwania łańcucha prototypów dla *wszystkich* umieszczonych w nim zmiennych, na ogół działa dużo wolniej niż zwykle bloki.

W JavaScriptcie nie istnieje lepszy mechanizm bezpośrednio zastępujący instrukcję `with`. Czasem lepszym rozwiązaniem jest proste wiązanie obiektu ze zmienną o krótkiej nazwie:

```
function status(info) {
    var w = new Widget();
    w.setBackground("blue");
    w.setForeground("white");
```

```
w.addText("Status: " + info);  
w.show();  
}
```

Działanie tej wersji jest dużo bardziej przewidywalne. Żadna z referencji nie jest tu zależna od zawartości obiektu `w`. Dlatego nawet jeśli w innym kodzie zmodyfikowany zostanie prototyp `Widget`, funkcja `status` nadal będzie działać w oczekiwany sposób:

```
status("connecting"); // Status: connecting  
Widget.prototype.info = "[[widget info]]";  
status("connected"); // Status: connected
```

W innych sytuacjach najlepszym rozwiązaniem jest bezpośrednie wiązanie zmiennych lokalnych z odpowiednimi właściwościami:

```
function f(x, y) {  
    var min = Math.min, round = Math.round, sqrt = Math.sqrt;  
    return min(round(x), sqrt(y));  
}
```

Także tu po wyeliminowaniu instrukcji `with` funkcja zaczyna działać w przewidywalny sposób:

```
Math.x = 0;  
Math.y = 0;  
f(2, 9); // 2
```

## Co warto zapamiętać?

- Unikaj stosowania instrukcji `with`.
- Stosuj zmienne o krótkich nazwach, jeśli chcesz wielokrotnie używać obiektu.
- Jawnie wiąż zmienne lokalne z właściwościami obiektów, zamiast niejawnie wiązać je za pomocą instrukcji `with`.

## Sposób 11. Poznaj domknięcia

Domknięcia (ang. *closure*) mogą być czymś nowym dla programistów używających języków, w których ta konstrukcja nie występuje. Początkowo domknięcia czasem wydają się skomplikowane. Wiedz jednak, że wysiłek włożony w ich opanowanie na pewno się opłaci.

Na szczęście tak naprawdę nie ma się czego bać. Opanowanie domknięć wymaga tylko zrozumienia trzech najważniejszych kwestii. Pierwsza z nich dotyczy tego, że JavaScript umożliwia wskazywanie zmiennych zdefiniowanych poza bieżącą funkcją:

```
function makeSandwich() {  
    var magicIngredient = "masło orzechowe";  
    function make(filling) {  
        return magicIngredient + " i " + filling;  
    }  
    return make("galaretka");  
}  
makeSandwich(); // "masło orzechowe i galaretka"
```

Zauważ, że w funkcji wewnętrznej `make` używana jest zmienna `magicIngredient`, zdefiniowana w funkcji zewnętrznej `makeSandwich`.

Druga kwestia związana jest z tym, że można używać zmiennych zdefiniowanych w funkcjach zewnętrznych nawet *po* zwróceniu wartości przez daną funkcję zewnętrzną! Jeśli wydaje Ci się to niemożliwe, pamiętaj, że w JavaScriptcie funkcje to obiekty pierwszoklasowe (zobacz Sposób 19.). To oznacza, że można zwrócić funkcję wewnętrzną i wywołać ją później:

```
function sandwichMaker() {  
    var magicIngredient = "masło orzechowe";  
    function make(filling) {  
        return magicIngredient + " i " + filling;  
    }  
    return make;  
}  
var f = sandwichMaker();  
f("galaretka"); // "masło orzechowe i galaretka"  
f("banany");    // "masło orzechowe i banany"  
f("pianki");    // "masło orzechowe i pianki"
```

Ten kod jest bardzo podobny jak w pierwszym przykładzie, jednak zamiast natychmiast zgłaszać wywołanie `make("galaretka")` w funkcji zewnętrznej, tu funkcja `sandwichMaker` zwraca samą funkcję `make`. Dlatego wartością wywołania `f` jest funkcja wewnętrzna `make`. Wywołanie `f` ostatecznie oznacza wywołanie funkcji `make`. Jednak choć funkcja `sandwichMaker` zwróciła już sterowanie, funkcja `make` zapamiętuje wartość zmiennej `magicIngredient`.

Jak to działa? Wartości reprezentujące funkcje w JavaScriptcie obejmują nie tylko kod potrzebny do wykonania ich w miejscu wywołania. Wewnętrznie przechowywane są też wszystkie zdefiniowane w zewnętrznym zasięgu zmienne, które mogą być używane w danej funkcji. Funkcje rejestrujące zmienne z zasięgu zewnętrznego to **domknięcia**. Funkcja `make` jest domknięciem, którego kod używa dwóch zmiennych zewnętrznych: `magicIngredient` i `filling`. Gdy funkcja `make` jest wywoływana, w jej kodzie można używać tych dwóch zmiennych, ponieważ są one przechowywane w domknięciu.

Funkcja ma dostęp do wszystkich zmiennych dostępnych w jej zasięgu — w tym do parametrów i zmiennych funkcji zewnętrznych. Można wykorzystać tę cechę do utworzenia ogólniejszej wersji funkcji `sandwichMaker`:

```
function sandwichMaker(magicIngredient) {
  function make(filling) {
    return magicIngredient + " i " + filling;
  }
  return make;
}
var hamAnd = sandwichMaker("szynka");
hamAnd("ser");           // "szynka i ser"
hamAnd("musztarda");     // "szynka i musztarda"
var turkeyAnd = sandwichMaker("indyk");
turkeyAnd("oscypek");    // "indyk i oscypek"
turkeyAnd("Provolone");  // "indyk i Provolone"
```

W tym przykładzie tworzone są dwie odrębne funkcje — `hamAnd` i `turkeyAnd`. Choć obie wykorzystują tę samą definicję funkcji `make`, są odrębnymi obiektami. W pierwszej funkcji wartością zmiennej `magicIngredient` jest `"szynka"`, natomiast w drugiej — `"indyk"`.

Domknięcia to jeden z najbardziej eleganckich i zwięzłych mechanizmów JavaScriptu. Jest on podstawą wielu przydatnych idiomów. JavaScript udostępnia nawet wygodniejszą składnię do tworzenia domknięć. Są to **wyrażenia funkcyjne**:

```
function sandwichMaker(magicIngredient) {
  return function(filling) {
    return magicIngredient + " i " + filling;
  };
}
```

Zauważ, że wyrażenie funkcyjne jest tu anonimowe. Nie trzeba podawać nazwy funkcji, ponieważ jest ona wykonywana tylko w celu uzyskania nowej wartości funkcji. Wyrażenie funkcyjne nie jest tu wywoływane lokalnie. Wyrażeniom funkcyjnym można jednak przypisywać nazwy (zobacz Sposób 14.).

Trzecia i ostatnia kwestia dotycząca domknięć dotyczy tego, że mogą one zmieniać wartości zmiennych zewnętrznych. Domknięcia przechowują *referencje* do zmiennych zewnętrznych, a nie kopie ich wartości. Dlatego modyfikacje są widoczne we wszystkich domknięciach, które mają dostęp do tych zmiennych. Ilustruje to prosty idiom **skrzynka** (ang. *box*). Przedstawia on obiekt przechowujący wewnętrzną wartość, którą można wczytać i zmodyfikować:

```
function box() {
  var val = undefined;
  return {
    set: function(newVal) { val = newVal; },
    get: function() { return val; },
    type: function() { return typeof val; }
  };
}
var b = box();
b.type(); // "undefined"
b.set(98.6);
b.get();  // 98.6
b.type(); // "number"
```



Ten przykład tworzy obiekt zawierający trzy domknięcia. Są nimi właściwości `set`, `get` i `type` tego obiektu. Każde z tych domknięć ma dostęp do zmiennej `val`. Domknięcie `set` ustawia wartość tej zmiennej, a w późniejszych wywołaniach `get` i `type` można zobaczyć efekt tej operacji.

### Co warto zapamiętać?

- Funkcje mogą używać zmiennych zdefiniowanych w zasięgach zewnętrznych.
- Domknięcia mogą działać także po zakończeniu pracy funkcji, w których je utworzono.
- Domknięcia wewnętrznie przechowują referencje do używanych zmiennych zewnętrznych i mogą zarówno wczytywać, jak i modyfikować wartości przechowywanych zmiennych.

## Sposób 12. Niejawne przenoszenie deklaracji zmiennych na początek bloku (czyli hoisting)

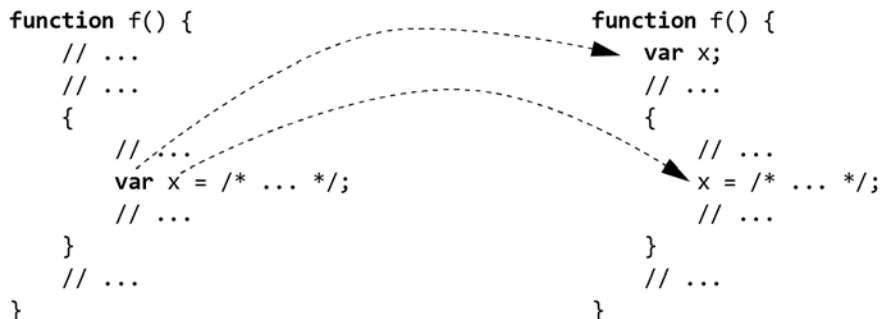
JavaScript obsługuje **zasięg leksykalny**. Z kilkoma wyjątkami referencja do zmiennej `foo` jest wiązana z najbliższym zasięgiem, w którym zadeklarowano `foo`. JavaScript nie obsługuje natomiast **zasięgu blokowego**. Definicje zmiennych nie mają zasięgu określonego na podstawie najbliższej instrukcji zewnętrznej lub bloku zewnętrznego; zasięg ustala się na podstawie funkcji zawierającej te zmienne.

Nieznajomość tych osobliwości JavaScriptu może prowadzić do powstawania trudnych do zauważenia błędów, takich jak pokazany poniżej:

```
function isWinner(player, others) {  
  var highest = 0;  
  for (var i = 0, n = others.length; i < n; i++) {  
    var player = others[i];  
    if (player.score > highest) {  
      highest = player.score;  
    }  
  }  
  return player.score > highest;  
}
```

Ten program na pozór deklaruje zmienną lokalną `player` w ciele pętli `for`. Jednak ponieważ zmienne w JavaScriptcie mają zasięg leksykalny, a nie blokowy, wewnętrzna deklaracja zmiennej `player` oznacza ponowną deklarację zmiennej, która już była dostępna w zasięgu, czyli parametru `player`. Każde wywołanie pętli powoduje zastąpienie wartości tej samej zmiennej. W efekcie w instrukcji `return` zmienna `player` ma wartość ostatniego elementu kolekcji `others`, zamiast pierwotnego argumentu `player` funkcji.

Deklaracje zmiennych w JavaScriptcie warto traktować jak operacje dwuczłowne, składające się z samej deklaracji i przypisania wartości. JavaScript niejawnie przenosi deklarację na początek funkcji zawierającej tę deklarację (ta technika to *hoisting*), a przypisanie pozostawia w pierwotnym miejscu. Oznacza to, że zmienna jest dostępna w zasięgu całej funkcji, ale wartość jest przypisywana do zmiennej dopiero w miejscu wystąpienia instrukcji `var`. Rysunek 2.2 przedstawia hoisting w formie wizualnej.



**Rysunek 2.2.** Hoisting zmiennych

Hoisting może też prowadzić do niejasności związanych z ponownymi deklaracjami zmiennych. Dozwolone jest wielokrotne deklarowanie tej samej zmiennej w zasięgu jednej funkcji. Często zdarza się to, gdy funkcja zawiera kilka pętli:

```

function trimSections(header, body, footer) {
  for (var i = 0, n = header.length; i < n; i++) {
    header[i] = header[i].trim();
  }
  for (var i = 0, n = body.length; i < n; i++) {
    body[i] = body[i].trim();
  }
  for (var i = 0, n = footer.length; i < n; i++) {
    footer[i] = footer[i].trim();
  }
}

```

Wydaje się, że w funkcji `trimSections` zadeklarowanych jest sześć zmiennych lokalnych (trzy o nazwie `i` oraz trzy o nazwie `n`). Jednak z powodu hoistingu tworzone są tylko dwie zmienne. Po hoistingu przedstawiona funkcja `trimSections` to odpowiednik jej następującej wersji:

```

function trimSections(header, body, footer) {
  var i, n;
  for (i = 0, n = header.length; i < n; i++) {
    header[i] = header[i].trim();
  }
  for (i = 0, n = body.length; i < n; i++) {
    body[i] = body[i].trim();
  }
}

```

```
for (i = 0, n = footer.length; i < n; i++) {  
    footer[i] = footer[i].trim();  
}  
}
```

Ponieważ ponowne deklaracje mogą powodować ustawianie różnych wartości zmiennych, niektórzy programiści wolą umieszczać wszystkie deklaracje `var` na początku funkcji. Działa to jak ręczny hoisting zmiennych i pozwala uniknąć wieloznaczności. Niezależnie od tego, czy stosujesz ten styl, ważna jest znajomość reguł określania zasięgu w JavaScriptcie (zarówno na potrzeby pisania, jak i czytania kodu).

Jeśli chodzi o brak zasięgu blokowego w JavaScriptcie, jedynym wyjątkiem są, jak się ciekawie składa, wyjątki. Konstrukcja `try...catch` wiąże przechwycony wyjątek ze zmienną, której zasięg jest ograniczony do danego bloku `catch`:

```
function test() {  
    var x = "var", result = [];  
    result.push(x);  
    try {  
        throw "exception";  
    } catch (x) {  
        x = "catch";  
    }  
    result.push(x);  
    return result;  
}  
test(); // ["var", "var"]
```

### Co warto zapamiętać?

- Deklaracje zmiennych z bloku są niejawnie przenoszone na początek funkcji zawierającej ten blok (ta technika to *hoisting*).
- Ponowne deklaracje zmiennej są traktowane jak instrukcje dotyczące tej samej zmiennej.
- Pomyśl o ręcznym hoistingu deklaracji zmiennych lokalnych w celu uniknięcia niejasności.

## Sposób 13. Stosuj wyrażenia IIFE do tworzenia zasięgu lokalnego

Jaki wynik zwróci poniższy (błędny!) program?

```
function wrapElements(a) {  
    var result = [], i, n;  
    for (i = 0, n = a.length; i < n; i++) {  
        result[i] = function() { return a[i]; };  
    }  
}
```

```

    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // ?

```

Programista mógł oczekiwać, że program zwróci wartość 10, jednak w rzeczywistości generowana jest wartość `undefined`.

Aby pojąć, dlaczego tak się dzieje, trzeba zrozumieć różnicę między wiązaniem a przypisywaniem. Wejście w dany zasięg w czasie wykonywania programu powoduje alokację miejsc w pamięci dla każdej zmiennej wiązanej w tym zasięgu. Funkcja `wrapElements` wiąże trzy zmienne lokalne: `result`, `i` oraz `n`. Dlatego w momencie wywołania tej funkcji przydzielane są miejsca na te trzy zmienne. W każdej iteracji pętli w jej ciele tworzone jest domknięcie w postaci funkcji zagnieżdżonej. Błąd w programie wynika z tego, że programista oczekuje, iż funkcja zapisze wartość zmiennej `i` z momentu utworzenia danej funkcji zagnieżdżonej. Jednak zachowywana jest *referencja* do zmiennej `i`. Ponieważ wartość `i` zmienia się po utworzeniu każdej funkcji zagnieżdżonej, funkcje te „widzą” ostateczną wartość `i`. Jest to ważna cecha domknięć:

*Domknięcia przechowują referencje do zmiennych zewnętrznych, a nie ich wartości.*

Dlatego wszystkie domknięcia utworzone przez funkcję `wrapElements` używają jednego współużytkowanego miejsca z wartością `i`, utworzonego przed uruchomieniem pętli. Ponieważ każda iteracja pętli zwiększa wartość `i` (do momentu wyjścia poza tablicę), w momencie wywołania domknięć z tablicy pobierana jest wartość o indeksie 5, dlatego zwracana wartość to `undefined`.

Zauważ, że funkcja `wrapElements` będzie działać dokładnie tak samo nawet wtedy, gdy deklaracje `var` znajdują się na początku pętli `for`:

```

function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
        result[i] = function() { return a[i]; };
    }
    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // undefined

```

Ta wersja wygląda jeszcze bardziej zwodniczo, ponieważ deklaracja `var` jest umieszczona w pętli. Jednak deklaracje zmiennych są przenoszone na początek funkcji, dlatego także tu dla zmiennej `i` przydzielone jest jedno miejsce.

Rozwiązanie polega na wymuszeniu utworzenia zasięgu lokalnego. W tym celu należy utworzyć funkcję zagnieżdżoną i od razu ją wywołać:

```
function wrapElements(a) {
  var result = [];
  for (var i = 0, n = a.length; i < n; i++) {
    (function() {
      var j = i;
      result[i] = function() { return a[j]; };
    })();
  }
  return result;
}
```

Ta technika to **natychmiast wywoływane wyrażenia funkcyjne** (ang. *Immediately Invoked Function Expression* — IIFE). Wyrażenia IIFE są niezastąpionym rozwiązaniem problemu braku zasięgu blokowego w JavaScriptcie. Inna możliwość to wiązanie zmiennej lokalnej z parametrem wyrażenia IIFE i przekazywanie wartości zmiennej jako argumentu.

```
function wrapElements(a) {
  var result = [];
  for (var i = 0, n = a.length; i < n; i++) {
    (function(j) {
      result[i] = function() { return a[j]; };
    })(i);
  }
  return result;
}
```

Zachowaj jednak ostrożność przy stosowaniu wyrażeń IIFE do tworzenia zasięgu lokalnego. Umieszczenie bloku w funkcji może spowodować subtelne zmiany w danym bloku. Po pierwsze, blok nie może wtedy zawierać żadnych instrukcji `break` lub `continue` wychodzących poza ten blok, ponieważ niedozwolone jest przechodzenie w ten sposób poza funkcję. Po drugie, jeśli w bloku używane jest słowo kluczowe `this` lub specjalna zmienna `arguments`, wyrażenie IIFE działa w inny sposób. Omówienie technik używania słów kluczowych `this` i `arguments` znajdziesz w rozdziale 3.

### Co warto zapamiętać?

- Zrozum różnicę między wiązaniem a przypisaniem.
- Domknięcia przechowują referencje do zmiennych zewnętrznych, a nie wartości tych zmiennych.
- Stosuj wyrażenia IIFE do tworzenia zasięgów lokalnych.
- Zwracaj uwagę na sytuacje, w których umieszczenie bloku w wyrażeniu IIFE może zmienić działanie bloku.

## Sposób 14. Uważaj na nieprzenośne określanie zasięgu nazwanych wyrażeń funkcyjnych

Funkcje JavaScript wyglądają tak samo niezależnie od miejsca, w którym się znajdują, jednak ich znaczenie zależy od kontekstu. Przyjrzyj się poniższemu fragmentowi kodu:

```
function double(x) { return x * 2; }
```

W zależności od miejsca występowania tego kodu może to być **deklaracja funkcji** lub **nazwane wyrażenie funkcyjne**. Deklaracja wygląda w standardowy sposób — definiowana jest funkcja, która zostaje związana ze zmienną w bieżącym zasięgu. Na przykład na najwyższym poziomie programu ta deklaracja tworzy funkcję globalną o nazwie `double`. Jednak ten sam kod funkcji można zastosować jako wyrażenie. Znaczy on wtedy coś zupełnie innego.

```
var f = function double(x) { return x * 2; };
```

Według specyfikacji ECMAScript ten kod wiąże funkcję ze zmienną `f`, a nie z `double`. Oczywiście do wyrażenia funkcyjnego nie trzeba przypisywać nazwy. Można zastosować anonimowe wyrażenie funkcyjne.

```
var f = function(x) { return x * 2; };
```

Oficjalna różnica między anonimowymi i nazwanymi wyrażeniami funkcyjnymi polega na tym, że nazwa wyrażeń nazwanych jest wiązana ze zmienną lokalną dostępną w tej funkcji. Można to wykorzystać do tworzenia rekurencyjnych wyrażeń funkcyjnych.

```
var f = function find(tree, key) {  
  if (!tree) {  
    return null;  
  }  
  if (tree.key === key) {  
    return tree.value;  
  }  
  return find(tree.left, key) ||  
    find(tree.right, key);  
};
```

Zauważ, że nazwa `find` jest dostępna tylko w zasięgu samej funkcji o tej nazwie. Nazwane wyrażenia funkcyjne (w odróżnieniu od zadeklarowanych funkcji) zewnętrznie nie mogą być używane z zastosowaniem wewnętrznej nazwy:

```
find(myTree, "foo"); // Błąd: nazwa find jest niezdefiniowana
```

Stosowanie nazwanych wyrażeń funkcyjnych przy rekurencji nie wydaje się specjalnie przydatne, ponieważ można wykorzystać także nazwę funkcji z zasięgu zewnętrznego:

```
var f = function(tree, key) {  
  if (!tree) {  
    return null;  
  }
```

```
    }
    if (tree.key === key) {
        return tree.value;
    }
    return f(tree.left, key) ||
        f(tree.right, key);
};
```

Inna możliwość to zastosowanie zwykłej deklaracji:

```
function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
        find(tree.right, key);
}
var f = find;
```

Nazwane wyrażenia funkcyjne są jednak naprawdę przydatne przy debugowaniu. Większość nowych środowisk JavaScriptu generuje ślad stosu dla obiektów `Error`. Nazwy wyrażeń funkcyjnych są stosowane w powiązanych z nimi wpisach w śladzie stosu. Debugery z narzędziami do inspekcji stosu zwykle wykorzystują nazwane wyrażenia funkcyjne w podobny sposób.

Niestety, nazwane wyrażenia funkcyjne są znane z tego, że powodują problemy z zasięgiem i kompatybilnością. Wynika to z nieszczęśliwej pomyłki w historii specyfikacji ECMAScript i błędów w popularnych silnikach JavaScriptu. Usterka w specyfikacji, występująca aż do wersji ES3, polegała na tym, że silniki JavaScript musiały reprezentować zasięg nazwanych wyrażeń funkcyjnych jako obiekt (podobnie jak w problematycznej konstrukcji `with`). Choć ten obiekt zasięgu obejmował tylko jedną właściwość (wiązącą nazwę funkcji z samą funkcją), dziedziczył też właściwości prototypu `Object.prototype`. Dlatego samo nazwanie wyrażenia funkcyjnego powodowało dodanie do zasięgu wszystkich właściwości prototypu `Object.prototype`. Efekty mogły być zaskakujące:

```
var constructor = function() { return null; };
var f = function f() {
    return constructor();
};
f(); // {} (w środowiskach zgodnych ze standardem ES3)
```

Wydaje się, że ten program powinien zwrócić wartość `null`, jednak w rzeczywistości zwraca nowy obiekt, ponieważ nazwane wyrażenie funkcyjne dziedziczy funkcję `Object.prototype.constructor` (czyli konstruktor z typu `Object`). Podobnie jak przy używaniu `with`, tak i tu na elementy dostępne w zasięgu wpływają dynamiczne zmiany prototypu `Object.prototype`. W jednej części programu właściwości prototypu `Object.prototype` mogą zostać dodane lub usunięte, co wpłynie na zmienne we wszystkich nazwanych wyrażeniach funkcyjnych.

Na szczęście w wersji ES5 ten błąd został naprawiony. Jednak w niektórych środowiskach JavaScriptu nadal stosowany jest dawny sposób określania zasięgu. Co gorsza, część środowisk jest jeszcze mniej zgodna ze standardami i wykorzystuje zasięgi w postaci obiektów *nawet dla anonimowych wyrażeń funkcyjnych!* W tych środowiskach nawet po usunięciu z przykładu nazwy wyrażenia funkcyjnego zwrócony zostanie obiekt zamiast oczekiwanej wartości `null`:

```
var constructor = function() { return null; };
var f = function() {
    return constructor();
};
f(); // {} (w środowiskach niezgodnych ze standardem)
```

Najlepszym sposobem na uniknięcie takich problemów w systemach, które zaśmiejają zasięg wyrażeń funkcyjnych obiektami, jest całkowita rezygnacja z dodawania nowych właściwości do prototypu `Object.prototype` i nieużywanie zmiennych lokalnych o nazwach odpowiadających standardowym właściwościom prototypu `Object.prototype`.

Następny błąd spotykany w popularnych silnikach JavaScriptu polega na hoistingu nazwanych wyrażeń funkcyjnych, tak jakby były deklaracjami. Oto przykład:

```
var f = function g() { return 17; };
g(); // 17 (w środowiskach niezgodnych ze standardem)
```

Warto podkreślić, że *nie* jest to działanie zgodne ze standardami. Co gorsza, niektóre środowiska JavaScriptu traktują obie funkcje `f` i `g` jak odrębne obiekty, co prowadzi do niepotrzebnego przydziału pamięci! Sensownym rozwiązaniem jest utworzenie zmiennej lokalnej o nazwie identycznej z nazwą wyrażenia funkcyjnego i przypisanie do tej zmiennej wartości `null`:

```
var f = function g() { return 17; };
var g = null;
```

Ponowna deklaracja tej zmiennej za pomocą słowa kluczowego `var` gwarantuje, że nazwa `g` zostanie związana nawet w środowiskach, które nie powodują błędnie hoistingu wyrażenia funkcyjnego, a ustawienie tej zmiennej na `null` prowadzi do odzyskania pamięci zajmowanej przez duplikat funkcji.

Uzasadnionym podsumowaniem jest stwierdzenie, że nazwane wyrażenia funkcyjne są zbyt problematyczne, aby warto było się nimi posługiwać. Mniej skrajnym podejściem jest stosowanie nazwanych wyrażeń funkcyjnych na potrzeby debugowania w trakcie pracy nad kodem. Następnie przed udostępnieniem kodu należy przekazać kod do preprocesora i przekształcić wszystkie wyrażenia funkcyjne na anonimową postać. Jedno jest jednak pewne — zawsze należy wiedzieć, na jakich platformach kod będzie wykonywany (zobacz Sposób 1.). Najgorsze, co można zrobić, to zaśmieszać kod sztuczkami, które w obsługiwanych platformach są niepotrzebne.



## Co warto zapamiętać?

- Stosuj nazwane wyrażenia funkcyjne do ulepszania śladów stosu w obiektach Error i debuggerach.
- Uważaj na zaśmiecanie zasięgu wyrażen funkcyjnych właściwościami prototypu `Object.prototype` w wersji ES3 standardu i nieprawidłowych środowiskach JavaScriptu.
- Uważaj na hoisting i powtórna alokację nazwanych wyrażen funkcyjnych w niewłaściwie działających środowiskach JavaScriptu.
- Rozważ rezygnację z nazwanych wyrażen funkcyjnych lub usuwanie ich przed udostępnieniem kodu.
- Jeśli udostępniasz kod w poprawnie zaimplementowanych środowiskach zgodnych ze standardem ES5, nie masz się o co martwić.

## Sposób 15. Uważaj na nieprzenośne określanie zasięgu lokalnych deklaracji funkcji w bloku

Deklaracje funkcji zagnieżdżonych to następny obszar, w którym występują problemy z zależnością od kontekstu. Może zaskoczyć Cię fakt, że nie istnieje standardowy sposób deklarowania funkcji w blokach lokalnych. Dozwolonym i często stosowanym rozwiązaniem jest zagnieżdżanie deklaracji funkcji na początku innej funkcji:

```
function f() { return "globalna"; }

function test(x) {
  function f() { return "lokalna"; }

  var result = [];
  if (x) {
    result.push(f());
  }
  result.push(f());
  return result;
}

test(true);    // ["lokalna", "lokalna"]
test(false);  // ["lokalna"]
```

Sytuacja zmienia się całkowicie po przeniesieniu `f` do bloku lokalnego:

```
function f() { return "globalna"; }

function test(x) {
  var result = [];
  if (x) {
    function f() { return "lokalna"; } // Deklaracja lokalna ograniczona do bloku
```

```

    result.push(f());
  }
  result.push(f());
  return result;
}

```

```

test(true);    // ?
test(false);   // ?

```

Możesz oczekiwać, że pierwsze wywołanie `test` zwróci tablicę `["lokalna", "globalna"]`, a drugie — tablicę `["globalna"]`, ponieważ wygląda tak, jakby wewnętrzna funkcja `f` była ograniczona do bloku `if`. Pamiętaj jednak, że w JavaScriptcie zasięg nie jest określany na podstawie bloków. Dlatego wewnętrzna funkcja `f` jest dostępna w zasięgu całego ciała funkcji `test`. Dobra druga próba odgadnięcia zwracanych wyników to `["lokalna", "lokalna"]` i `["lokalna"]`. Niektóre środowiska JavaScriptu działają właśnie w ten sposób. Jednak nie wszystkie! Inne *warunkowo* wiążą wewnętrzną funkcję `f` w czasie wykonywania programu w zależności od tego, czy zawierający ją blok zostanie wykonany. Nie tylko utrudnia to zrozumienie kodu, ale też prowadzi do niskiej wydajności (przypomina to nieco sytuację z instrukcją `with`).

Co na ten temat można wyczytać w standardzie ECMAScript? Co zaskakujące, prawie nic. Do wersji ES5 w standardzie nawet nie uwzględniano występowania lokalnych deklaracji funkcji w bloku. Według oficjalnej specyfikacji deklaracje funkcji występują tylko w programach lub na najwyższym poziomie w innych funkcjach. Standard ES5 zaleca nawet zgłaszanie ostrzeżenia lub błędu po wykryciu deklaracji funkcji w niestandardowych kontekstach. Popularne implementacje JavaScriptu działające w trybie *strict* zgłaszają takie deklaracje jako błąd. Program w trybie *strict* zawierający lokalną deklarację funkcji w bloku zgłosi błąd składni. Pomaga to wykryć nieprzenośny kod i powoduje, że w przyszłych standardach łatwiej będzie opracować bardziej sensowne i przenośne działanie lokalnych deklaracji funkcji w blokach.

Do tego czasu najlepszym sposobem na pisanie przenośnych funkcji jest unikanie umieszczania deklaracji funkcji w blokach lokalnych lub instrukcjach podrzędnych. Jeśli chcesz dodać deklarację funkcji zagnieżdżonej, umieść ją na najwyższym poziomie funkcji nadrzędnej, tak jak w pierwszej wersji kodu z tego sposobu. Jeżeli jednak chcesz warunkowo wybierać funkcje, to najlepiej zastosuj deklaracje `var` i wyrażenia funkcyjne:

```

function f() { return "globalna"; }

function test(x) {
  var g = f, result = [];
  if (x) {
    g = function() { return "lokalna"; }
  }
  result.push(g());
}

```

```
    result.push(g());  
    return result;  
}
```

To eliminuje niejednoznaczność związaną z zasięgiem zmiennej wewnętrznej (tu używana jest nazwa `g`). Zmienna jest bezwarunkowo wiązana jako zmienna lokalna i jedynie przypisanie jest warunkowe. Efekt to jednoznaczny i w pełni przenośny kod.

### Co warto zapamiętać?

- Zawsze umieszczaj deklaracje funkcji na najwyższym poziomie programu lub funkcji. W ten sposób unikniesz nieprzenośnego kodu.
- Stosuj deklaracje `var` z warunkowym przypisywaniem zamiast warunkowych deklaracji funkcji.

## Sposób 16. Unikaj tworzenia zmiennych lokalnych za pomocą funkcji eval

Funkcja `eval` w JavaScriptcie daje bardzo dużo możliwości i swobody. Narzędzia tego rodzaju łatwo jest zastosować w niewłaściwy sposób, dlatego warto dobrze je zrozumieć. Jedną z najprostszych dróg do problemów z funkcją `eval` jest użycie jej w ten sposób, że wpływa na zasięg.

W wywołaniu funkcji `eval` jej argument jest interpretowany jako program w JavaScriptcie. Ten program zostaje uruchomiony w lokalnym zasięgu programu wywołującego. Zmienne globalne programu zagnieżdżonego są tworzone jako zmienne lokalne programu wywołującego:

```
function test(x) {  
    eval ("var y = x;"); // Wiązanie dynamiczne  
    return y;  
}  
test("Witaj"); // "Witaj"
```

Ten przykład wygląda zrozumiale, jednak działa nieco inaczej, niż gdyby deklarację `var` umieszczono bezpośrednio w ciele funkcji `test`. Deklaracja `var` jest tu przetwarzana dopiero w momencie wywołania funkcji `eval`. Umieszczenie funkcji `eval` w bloku warunkowym powoduje, że jej zmienne znajdą się w zasięgu tylko wtedy, gdy ten blok zostanie wykonany:

```
var y = "globalna";  
function test(x) {  
    if (x) {  
        eval("var y = 'lokalna'"); // Wiązanie dynamiczne  
    }  
    return y;  
}
```

```
test(true);    // "lokalna"
test(false);  // "globalna"
```

Opieranie decyzji związanych z zasięgiem na dynamicznych operacjach to prawie zawsze zły pomysł. W takiej sytuacji proste ustalenie, w jaki sposób wiązana jest zmienna, wymaga zrozumienia szczegółów działania programu. Sprawia to trudność zwłaszcza wtedy, gdy kod źródłowy przekazywany do funkcji `eval` nie jest nawet zdefiniowany lokalnie:

```
var y = "globalna";
function test(src) {
    eval(src); // Może powodować dynamiczne wiązanie zmiennych
    return y;
}
test("var y = 'lokalna';"); // "lokalna"
test("var z = 'lokalna';"); // "globalna"
```

Ten kod jest narażony na błędy i niebezpieczny. Umożliwia zewnętrznym programom wywołującym zmianę elementów z wewnętrznego zasięgu funkcji `test`. Oczekiwanie, że funkcja `eval` zmodyfikuje zasięg, w którym się znajduje, jest też niezgodne z trybem `strict` ze standardu ES5. W tym trybie funkcja `eval` powinna działać w zasięgu zagnieżdżonym, aby zapobiec zaśmiecaniu zasięgu. Prosty sposób na upewnienie się, że funkcja `eval` nie wpłynie na zewnętrzny zasięg, to wywołanie jej w jawnie zagnieżdżonym zasięgu:

```
var y = "globalna";
function test(src) {
    (function() { eval(src); })();
    return y;
}
test("var y = 'lokalna';"); // "globalna"
test("var z = 'lokalna';"); // "globalna"
```

### Co warto zapamiętać?

- Unikaj tworzenia za pomocą funkcji `eval` zmiennych, które zaśmiecają zasięg programu wywołującego.
- Jeśli kod przekazany do funkcji `eval` może tworzyć zmienne globalne, umieść wywołanie w funkcji zagnieżdżonej. Dzięki temu unikniesz zaśmiecania zasięgu.

### Sposób 17. Przedkładaj pośrednie wywołania `eval` nad bezpośrednie wywołania tej funkcji

Z funkcją `eval` związana jest pewna tajemnica — `eval` to coś więcej niż funkcja.

Większość funkcji ma tylko dostęp do zasięgu, w którym zostały zdefiniowane. Jednak `eval` ma dostęp do pełnego zasięgu z *miejsca wywołania*. Daje to takie

możliwości, że gdy twórcy kompilatorów po raz pierwszy próbowali zoptymalizować JavaScript, odkryli, że eval utrudnia wydajną obsługę wywołań funkcji, ponieważ w każdym wywołaniu funkcji trzeba było udostępniać zasięg w czasie wykonywania programu na wypadek, gdyby używana była właśnie funkcja eval.

W ramach kompromisu zmodyfikowano standard języka i uwzględniono w nim dwa różne sposoby wywoływania funkcji eval. Wywołanie z identyfikatorem eval jest uznawane za wywołanie bezpośrednie:

```
var x = "globalna";
function test() {
    var x = "lokalna";
    return eval("x"); // Bezpośrednie wywołanie eval
}
test(); // "lokalna"
```

W tym podejściu kompilatory muszą zapewnić wykonywanemu programowi kompletny dostęp do lokalnego zasięgu programu wywołującego. Inny rodzaj wywołań funkcji eval to wywołania pośrednie. W tym przypadku argument jest przetwarzany w zasięgu globalnym. Jeśli na przykład powiążesz funkcję eval z nazwą zmiennej i wywołasz tę funkcję za pomocą tej nazwy, kod nie będzie miał dostępu do zasięgu lokalnego:

```
var x = "globalna";
function test() {
    var x = "lokalna";
    var f = eval;
    return f("x"); // Wywołanie pośrednie
}
test(); // "globalna"
```

Definicja bezpośrednich wywołań funkcji eval jest opisana w standardzie ECMAScript w niejednoznaczny sposób. W praktyce jedyna składnia, która prowadzi do bezpośredniego wywołania funkcji eval, to zmienna o nazwie eval; może być ona umieszczona w dowolnej liczbie nawiasów. Zwięzły zapis pośredniego wywołania funkcji eval obejmuje operator sekwencji wyrażenia (.) i pozbawiony znaczenia literal liczbowy:

```
(0,eval)(src);
```

Jak zadziała to dziwnie wyglądające wywołanie funkcji? Literal liczbowy 0 jest przetwarzany, ale jego wartość zostaje zignorowana. W efekcie sekwencja wyrażeń zwraca funkcję eval. Tak więc zapis (0,eval) działa prawie dokładnie tak samo jak sam identyfikator eval, ale z jedną istotną różnicą — całe wyrażenie jest traktowane jak pośrednie wywołanie funkcji eval.

Możliwości bezpośredniego wywołania eval mogą łatwo zostać wykorzystane w niewłaściwy sposób. Na przykład przetwarzanie źródłowego łańcucha znaków pobranego z sieci naraża wewnętrzne mechanizmy programu na atak ze

strony niezaufanych jednostek. W sposobie 16. opisuję zagrożenia związane z dynamicznym tworzeniem zmiennych lokalnych przez funkcję `eval`. Te problemy mogą wystąpić tylko przy bezpośrednim wywołaniu funkcji `eval`. Ponadto bezpośrednie wywołania `eval` powodują znaczny spadek wydajności. Zwykle bezpośrednie wywołania `eval` powodują, że zawierająca je funkcja i *wszystkie kolejne funkcje zewnętrzne aż do najwyższego poziomu programu* będą działać znacznie wolniej.

Czasem są powody do stosowania bezpośrednich wywołań `eval`. Jednak jeśli nie istnieje wyraźna potrzeba korzystania z zasięgu lokalnego, stosuj mniej problematyczne i mniej kosztowne ze względu na wydajność pośrednie wywołania `eval`.

### Co warto zapamiętać?

- Umieszczaj wywołania `eval` w sekwencjach wyrażeń razem z pomijanym literałem, aby wymusić pośrednie wywołanie tej funkcji.
- Zawsze, gdy jest to możliwe, przedkładaj pośrednie wywołania `eval` nad wywołania bezpośrednie.

# 3

## Korzystanie z funkcji

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

### Sposób 18. Różnice między wywołaniami funkcji, metod i konstruktorów

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

Najprostszy sposób to wywołanie funkcji:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

### Co warto zapamiętać?

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

## Sposób 19. Funkcje wyższego poziomu

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Inna część programu generuje łańcuch znaków obejmujący cyfry:

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

W jeszcze innym fragmencie program tworzy łańcuch z losowych znaków:

```
var random = "";

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

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

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

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

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

```
var alphabet = buildString(26, function(i) {  
  return String.fromCharCode(aIndex + i);  
});  
alphabet; // "abcdefghijklmnopqrstuvwxyz"  
  
var digits = buildString(10, function(i) { return i; });  
digits; // "0123456789"  
  
var random = buildString(8, function() {  
  return String.fromCharCode(Math.floor(Math.random() * 26)  
    + aIndex);  
});  
random; // "ltvisfr" (wynik za każdym razem jest inny)
```

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

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

### Co warto zapamiętać?

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

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

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

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

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

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

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

działa podobnie jak wywołanie bezpośrednie:

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

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

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

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

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

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

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

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

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

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

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

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

## Co warto zapamiętać?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Metodę `append` można wywołać z dowolną liczbą argumentów:

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

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

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

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

### Co warto zapamiętać?

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



## Sposób 22. Stosuj słowo kluczowe arguments do tworzenia funkcji wariadycznych

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

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

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

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

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

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

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

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

### Co warto zapamiętać?

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

### Sposób 23. Nigdy nie modyfikuj obiektu `arguments`

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

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

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

Jednak działanie tej funkcji jest dalekie od oczekiwanego:

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

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

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

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

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

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

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

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

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

Teraz metoda `callMethod` działa zgodnie z oczekiwaniami:

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

## Co warto zapamiętać?

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

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

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

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

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

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

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

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

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

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

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

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

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

### Co warto zapamiętać?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### Co warto zapamiętać?

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

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

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

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

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

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

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

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

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

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

### Co warto zapamiętać?

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

## Sposób 27. Wybieraj domknięcia zamiast łańcuchów znaków do hermetyzowania kodu

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



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

Przyjrzyj się prostej funkcji wielokrotnie powtarzającej określoną przez użytkownika operację:

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

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

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

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

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

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

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

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

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

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

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

### Co warto zapamiętać?

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

### Sposób 28. Unikaj stosowania metody `toString` funkcji

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

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

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

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

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

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

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

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

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

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

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

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

### Co warto zapamiętać?

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

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

## Sposób 29. Unikaj niestandardowych właściwości przeznaczonych do inspekcji stosu

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

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

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

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

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

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

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

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

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

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

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

Dla prostych stosów wywołań funkcja `getCallStack` działa poprawnie.

```
function f1() {  
    return getCallStack();  
}  
  
function f2() {  
    return f1();  
}  
  
var trace = f2();  
trace; // [f1, f2]
```

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

```
function f(n) {  
    return n === 0 ? getCallStack() : f(n - 1);  
}  
  
var trace = f(1); // Pętla nieskończona
```

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

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

```
function f() {  
    "use strict";  
    return f.caller;  
}  
  
f(); // Błąd — nie można używać właściwości caller funkcji w trybie strict
```

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

### Co warto zapamiętać?

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

# 4

## Obiekty i prototypy

Obiekt to podstawowa struktura danych w JavaScriptcie. Ogólnie można stwierdzić, że obiekt reprezentuje tabelę łączącą łańcuchy znaków z wartościami. Jednak gdy przyjrzyś się sprawie dokładniej, zobaczysz, że obiekty obejmują rozbudowane mechanizmy.

JavaScript, podobnie jak wiele języków obiektowych, obsługuje **dziedziczenie implementacji**. Pozwala to ponownie wykorzystać kod lub dane za pomocą mechanizmu dynamicznej delegacji. Jednak (inaczej niż w wielu konwencjonalnych językach) dziedziczenie w JavaScriptcie jest oparte na prototypach, a nie na klasach. Dla wielu programistów JavaScript to pierwszy poznawany język obiektowy bez klas.

W licznych językach każdy obiekt jest egzemplarzem określonej klasy. Kod klasy jest współużytkowany przez wszystkie jej egzemplarze. Natomiast w JavaScriptcie nie ma wbudowanego mechanizmu klas. Obiekty dziedziczą po innych obiektach. Każdy obiekt jest powiązany z innym obiektem — jego **prototypem**. Prototypów w niektórych aspektach używa się inaczej niż klas, choć dostępnych jest wiele rozwiązań z tradycyjnych języków obiektowych.

### Sposób 30. Różnice między instrukcjami `prototype`, `getPrototypeOf` i `__proto__`

Dla prototypów istnieją trzy różne, ale powiązane ze sobą akcesory. Nazwa każdego z nich oparta jest na słowie *prototype*. Ta nieszczęśliwa zbieżność często prowadzi do wątpliwości. Przejdźmy od razu do rzeczy.

- Instrukcja `C.prototype` służy do określenia prototypu obiektów tworzonych za pomocą wywołań `new C()`.
- Instrukcja `Object.getPrototypeOf(obj)` to standardowy mechanizm ze specyfikacji ES5 pozwalający pobrać prototyp obiektu `obj`.

- Instrukcja `obj.__proto__` to niestandardowy mechanizm pozwalający pobrać prototyp obiektu `obj`.

Aby zrozumieć każdą z tych instrukcji, przyjrzyj się typowej definicji typu danych JavaScriptu. Konstruktor obiektu `User` jest wywoływany za pomocą operatora `new` i przyjmuje nazwę użytkownika oraz skrót hasła. Te dane są zapisywane w tworzonym obiekcie.

```
function User(name, passwordHash) {  
    this.name = name;  
    this.passwordHash = passwordHash;  
}  
  
User.prototype.toString = function() {  
    return "[User " + this.name + "]";  
};  
  
User.prototype.checkPassword = function(password) {  
    return hash(password) === this.passwordHash;  
};  
  
var u = new User("sfalken",  
    "0ef33ae791068ec64b502d6cb0191387");
```

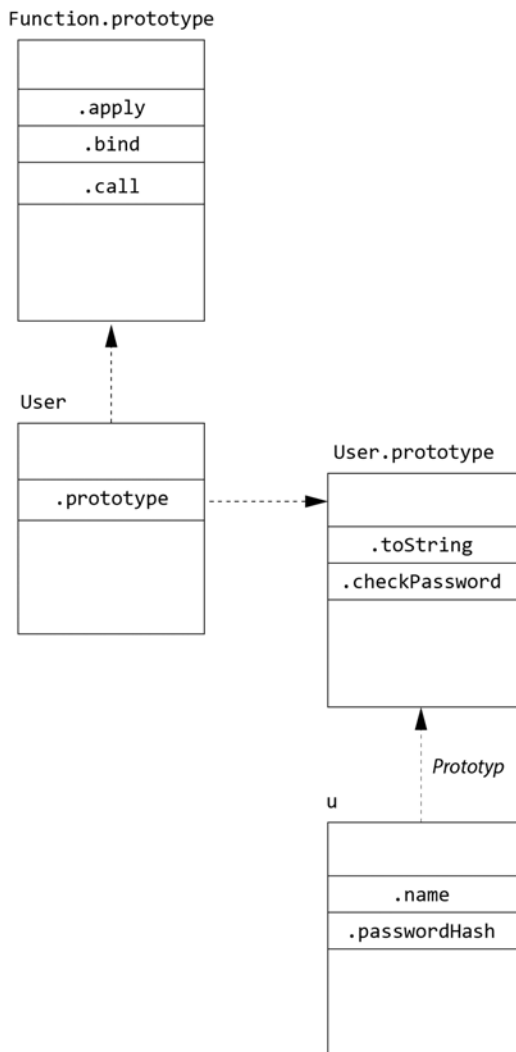
Funkcja `User` ma domyślną właściwość `prototype`, obejmującą obiekt, który początkowo jest w zasadzie pusty. W tym przykładzie do obiektu `User.prototype` dodawane są dwie metody: `toString` i `checkPassword`. Gdy stworzysz egzemplarz typu `User` za pomocą operatora `new`, dla uzyskanego obiektu `u` jako prototyp automatycznie ustawiany jest obiekt z właściwości `User.prototype`. Diagram z tymi obiektami znajdziesz na rysunku 4.1.

Zwróć uwagę na strzałkę łączącą obiekt `u` z prototypem `User.prototype`. Reprezentuje ona relację dziedziczenia. Wyszukiwanie właściwości rozpoczyna się od sprawdzenia *własnych właściwości* obiektu. Na przykład instrukcje `u.name` i `u.passwordHash` zwracają bieżące wartości właściwości obiektu `u`. Jeśli właściwości nie są bezpośrednio dostępne w `u`, są wyszukiwane w prototypie tego obiektu. Na przykład instrukcja `u.checkPassword` zwraca metodę zapisaną w prototypie `User.prototype`.

To prowadzi do następnego elementu z listy. Właściwość `prototype` konstruktora służy do określania prototypu nowych egzemplarzy. Funkcja `Object.getPrototypeOf()` ze standardu ES5 pozwala pobrać prototyp istniejącego obiektu. Tak więc po utworzeniu obiektu `u` w przedstawionym przykładzie można przeprowadzić następujący test:

```
Object.getPrototypeOf(u) === User.prototype; // true
```





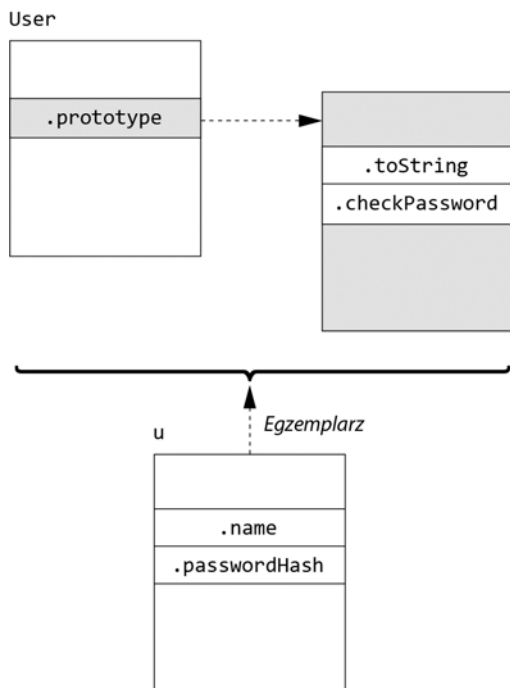
**Rysunek 4.1.** Relacje prototypu z konstruktorem i egzemplarzem typu User

W niektórych środowiskach dostępny jest niestandardowy mechanizm pobierania prototypów obiektów — właściwość `__proto__`. Jest to przydatne jako tymczasowe rozwiązanie w środowiskach, które nie obsługują instrukcji `Object.getPrototypeOf` ze standardu ES5. W takich środowiskach można przeprowadzić podobny test:

```
u.__proto__ === User.prototype; // true
```

Ostatnia uwaga na temat relacji z prototypem — wielu programistów JavaScriptu nazwałoby User *klasą*, choć nie zawiera ona prawie nic oprócz funkcji. Klasy w JavaScriptcie to połączenie konstruktora (User) i prototypu używanego do współużytkowania metod między egzemplarzami klasy (User.prototype).

Rysunek 4.2 przedstawia wygodny sposób myślenia o klasie `User`. Funkcja `User` zapewnia publiczny konstruktor klasy, a `User.prototype` to wewnętrzna implementacja metod współużytkowanych przez egzemplarze. W zwykłych zastosowaniach jednostek `User` i `u` nie trzeba bezpośrednio używać prototypu.



**Rysunek 4.2.** Schematyczna ilustracja „klasy” `User`

### Co warto zapamiętać?

- Instrukcja `C.prototype` pozwala określić prototyp obiektów tworzonych za pomocą wywołań `new C()`.
- `Object.getPrototypeOf(obj)` to standardowa funkcja ze specyfikacji ES5 służąca do pobierania prototypów obiektów.
- Instrukcja `obj.__proto__` to niestandardowy mechanizm pobierania prototypów obiektów.
- Klasa to wzorzec projektowy obejmujący konstruktor i powiązany z nim prototyp.

## Sposób 31. Stosuj instrukcję `Object.getPrototypeOf` zamiast `__proto__`

W standardzie ES5 wprowadzono instrukcję `Object.getPrototypeOf`. Jest to standardowy interfejs API do pobierania prototypów obiektów. Jednak przed jego dodaniem w wielu silnikach JavaScriptu dostępna była właściwość `__proto__` mająca to samo przeznaczenie. Jednak nie wszystkie środowiska JavaScriptu obsługują tę właściwość. Poza tym w środowiskach udostępniających tę właściwość nie działa ona w identyczny sposób. Środowiska różnią się na przykład w zakresie obsługi obiektów z prototypem `null`. W niektórych środowiskach właściwość `__proto__` jest dziedziczona po `Object.prototype`, dlatego obiekt z prototypem `null` nie ma tej właściwości.

```
var empty = Object.create(null); // Obiekt bez prototypu
"__proto__" in empty; // false (w niektórych środowiskach)
```

W innych środowiskach właściwość `__proto__` zawsze jest obsługiwana w specjalny sposób — niezależnie od stanu obiektu.

```
var empty = Object.create(null); // Obiekt bez prototypu
"__proto__" in empty; // true (w niektórych środowiskach)
```

Gdy metoda `Object.getPrototypeOf` jest dostępna, stanowi bardziej standardowy i przenośny sposób pobierania prototypów. Ponadto właściwość `__proto__` powoduje liczne błędy, ponieważ jest dołączana do wszystkich obiektów (zobacz Sposób 45.). W przyszłości w silnikach JavaScriptu obsługujących tę właściwość może pojawić się opcja wyłączania jej w celu uniknięcia błędów. Stosowanie metody `Object.getPrototypeOf` gwarantuje, że kod będzie działał także wtedy, gdy właściwość `__proto__` jest wyłączona.

W środowiskach JavaScriptu, które nie udostępniają opisanego interfejsu API ze standardu ES5, można łatwo zaimplementować go z wykorzystaniem właściwości `__proto__`.

```
if (typeof Object.getPrototypeOf === "undefined") {
  Object.getPrototypeOf = function(obj) {
    var t = typeof obj;
    if (!obj || (t !== "object" && t !== "function")) {
      throw new TypeError("To nie obiekt");
    }
    return obj.__proto__;
  };
}
```

Tę implementację można bezpiecznie stosować w środowiskach zgodnych ze standardem ES5, ponieważ nie dodaje ona przedstawionej funkcji, jeśli metoda `Object.getPrototypeOf` już istnieje.

### Co warto zapamiętać?

- Stosuj zgodną ze standardami metodę `Object.getPrototypeOf` zamiast nie-standardowej właściwości `__proto__`.
- W środowiskach nieobsługujących standardu ES5, ale udostępniających właściwość `__proto__`, implementuj własną metodę `Object.getPrototypeOf`.

### Sposób 32. Nigdy nie modyfikuj właściwości `__proto__`

Specjalna właściwość `__proto__` daje dodatkową możliwość, której nie zapewnia metoda `Object.getPrototypeOf`. Chodzi tu o możliwość *modyfikowania* wiązania obiektu z prototypem. Choć na pozór jest to nieszkodliwe (w końcu to tylko jedna z wielu właściwości, prawda?), takie działanie może mieć poważne skutki, dlatego należy go unikać. Najbardziej oczywistym powodem do unikania modyfikowania właściwości `__proto__` są kwestie związane z przenośnością kodu. Ponieważ nie wszystkie systemy umożliwiają zmianę prototypu obiektu, nie da się napisać przenośnego kodu, który wykonuje taką operację.

Innym powodem unikania modyfikowania właściwości `__proto__` jest wydajność. We wszystkich nowych silnikach JavaScriptu zadania pobierania i ustawiania właściwości obiektów są wysoce zoptymalizowane, ponieważ należą do najczęściej wykonywanych operacji w programach pisanych w tym języku. Te optymalizacje są oparte na znajomości struktury obiektu przez silnik. Zmiana wewnętrznej struktury obiektu (na przykład w wyniku dodania lub usunięcia jego właściwości lub właściwości jednego z prototypowych obiektów) powoduje, że niektóre optymalizacje przestają działać. Modyfikacja właściwości `__proto__` zmienia strukturę dziedziczenia. Jest to jedna z najbardziej destrukcyjnych zmian. Może ona doprowadzić do unieważnienia znacznie większej liczby optymalizacji niż modyfikacje zwykłych właściwości.

Jednak najważniejszym argumentem na rzecz unikania modyfikowania właściwości `__proto__` jest to, że w ten sposób można zapewnić przewidywalne działanie kodu. Łańcuch prototypów obiektu definiuje jego działanie na podstawie zbioru dostępnych właściwości i ich wartości. Modyfikacja wiązania obiektu z prototypem jest jak przeszczep mózgu i zmienia całą hierarchię dziedziczenia obiektu. Można wyobrazić sobie wyjątkowe sytuacje, w których takie rozwiązanie jest przydatne. Jednak prawie zawsze hierarchia dziedziczenia powinna pozostawać stabilna.

Do tworzenia nowych obiektów z niestandardowym wiązaniem z prototypem służy metoda `Object.create` ze specyfikacji ES5. W środowiskach, które nie obsługują standardu ES5, można zastosować opisaną w sposobie 33. przenośną implementację metody `Object.create`, nieopartą na właściwości `__proto__`.

## Co warto zapamiętać?

- Nigdy nie modyfikuj właściwości `__proto__` obiektu.
- Do tworzenia niestandardowych prototypów nowych obiektów używaj metody `Object.create`.

## Sposób 33. Uniezależnianie konstruktorów od instrukcji new

Gdy stworzysz konstruktor (taki jak funkcja `User` w sposobie 30.), musisz przyjąć, że wywołujące go osoby będą pamiętały o użyciu operatora `new`. Zauważ, że funkcja zakłada, iż odbiorca to nowy obiekt.

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

Jeśli osoba wywołująca konstruktor zapomni o słowie kluczowym `new`, odbiorcą funkcji będzie obiekt globalny.

```
var u = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");  
u; // undefined  
this.name; // "baravelli"  
this.passwordHash; // "d8b74df393528d51cd19980ae0aa028e"
```

To wywołanie funkcji nie tylko zwraca bezużyteczną wartość `undefined`, ale też, co bardzo szkodliwe, tworzy zmienne globalne `name` i `passwordHash` (lub modyfikuje je, jeśli już istnieją).

Jeżeli dla funkcji `User` używany jest tryb `strict` ze standardu ES5, odbiorcą domyślnie jest wartość `undefined`.

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

```
var u = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");  
// Błąd: obiekt this jest niezdefiniowany
```

W tej sytuacji błędne wywołanie natychmiast prowadzi do błędu. Pierwszy wiersz funkcji `User` próbuje przypisać wartość do `this.name`, co powoduje błąd `TypeError`. Dlatego konstruktor działający w trybie `strict` pozwala użytkownikowi szybko wykryć błąd i go naprawić.

Jednak w obu sytuacjach funkcja `User` jest podatna na błędy. Gdy używa się jej z operatorem `new`, działa zgodnie z oczekiwaniami, jeśli jednak wywołać ją jak zwykłą funkcję, pojawiają się problemy. Bardziej niezawodnym rozwiązaniem jest udostępnienie funkcji działającej jak konstruktor niezależnie od sposobu

jej wywołania. Łatwy sposób na zaimplementowanie tego rozwiązania to sprawdzanie, czy odbiorcą jest egzemplarz typu `User`.

```
function User(name, passwordHash) {
  if (!(this instanceof User)) {
    return new User(name, passwordHash);
  }
  this.name = name;
  this.passwordHash = passwordHash;
}
```

W tym podejściu wynik wywołania funkcji `User` to obiekt dziedziczący po prototypie `User.prototype`. Jest tak niezależnie od tego, czy obiekt został wywołany jako funkcja, czy jako konstruktor.

```
var x = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
var y = new User("baravelli",
  "d8b74df393528d51cd19980ae0aa028e");
x instanceof User; // true
y instanceof User; // true
```

Wadą tego wzorca jest to, że konieczne jest dodatkowe wywołanie funkcji, co zmniejsza wydajność. Trudno jest też stosować to rozwiązanie dla funkcji o różnej liczbie argumentów (zobacz sposoby 21. i 22.), ponieważ nie istnieje prosty odpowiednik metody `apply` pozwalający wywoływać takie funkcje jako konstruktory. W poniższym, bardziej wymyślnym podejściu używana jest metoda `Object.create` ze standardu ES5.

```
function User(name, passwordHash) {
  var self = this instanceof User
    ? this
    : Object.create(User.prototype);
  self.name = name;
  self.passwordHash = passwordHash;
  return self;
}
```

Metoda `Object.create` przyjmuje prototyp i zwraca nowy dziedziczący po nim obiekt. Dlatego gdy ta wersja funkcji `User` zostanie wywołana jak zwykła funkcja, zwróci nowy obiekt dziedziczący po prototypie `User.prototype`, mający zainicjowane właściwości `name` i `passwordHash`.

Choć metoda `Object.create` jest dostępna tylko w standardzie ES5, w starszych środowiskach można opracować podobne rozwiązanie. Wymaga to utworzenia lokalnego konstruktora i wywoływania go za pomocą operatora `new`.

```
if (typeof Object.create === "undefined") {
  Object.create = function(prototype) {
    function C() {}
    C.prototype = prototype;
    return new C();
  };
}
```

Zauważ, że ten kod odpowiada tylko jednoargumentowej wersji metody `Object.create`. Ta metoda przyjmuje także opcjonalny drugi argument, który opisuje zestaw deskryptorów właściwości nowego obiektu.

Co się stanie, gdy użytkownik wywoła nową wersję funkcji `User` za pomocą operatora `new`? Dzięki wzorcowi **przesłaniania konstruktora** kod zadziała tak jak przy normalnym wywołaniu funkcji. Dzieje się tak, ponieważ JavaScript umożliwia zastąpienie wyniku wyrażenia `new` wynikiem jawnej instrukcji `return` z konstruktora. Gdy funkcja `User` zwraca obiekt `self`, wynikiem wyrażenia `new` staje się właśnie `self`. Może to być inny obiekt niż `this`.

Zabezpieczanie konstruktora przed błędnym użyciem nie zawsze jest warte zachodu — zwłaszcza gdy konstruktora używasz tylko lokalnie. Należy jednak pamiętać, że niewłaściwe wywołanie konstruktora może prowadzić do problemów. Warto przynajmniej udokumentować, że konstruktor należy wywoływać za pomocą operatora `new` (zwłaszcza gdy jest on używany w rozbudowanym rozwiązaniu lub jest częścią biblioteki współużytkowanej).

### Co warto zapamiętać?

- Spraw, aby konstruktor był niezależny od składni w wywołującym go kodzie. W tym celu należy w samym konstruktorze zastosować operator `new` lub metodę `Object.create`.
- Jednoznacznie dokumentuj funkcje, które należy wywoływać za pomocą operatora `new`.

## Sposób 34. Umieszczaj metody w prototypach

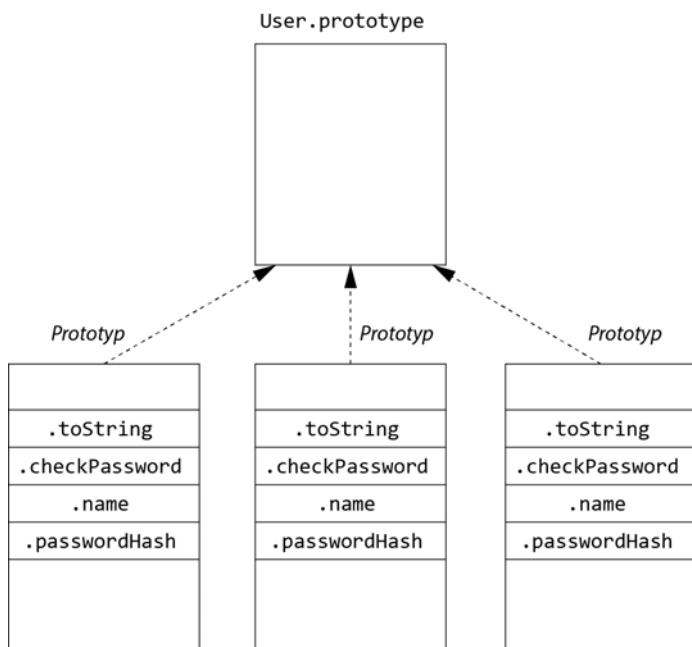
Możliwe jest programowanie w JavaScriptcie bez używania prototypów. Klasę `User` ze sposobu 30. można zaimplementować bez definiowania żadnych specjalnych elementów w jej prototypie.

```
function User(name, passwordHash) {  
  this.name = name;  
  this.passwordHash = passwordHash;  
  this.toString = function() {  
    return "[Użytkownik " + this.name + "]";  
  };  
  this.checkPassword = function(password) {  
    return hash(password) === this.passwordHash;  
  };  
}
```

W większości zastosowań ta klasa działa podobnie jak jej pierwotna wersja. Jednak gdy tworzonych jest kilka egzemplarzy typu `User`, pojawia się istotna różnica.

```
var u1 = new User(/* ... */);
var u2 = new User(/* ... */);
var u3 = new User(/* ... */);
```

Rysunek 4.3 pokazuje, jak wyglądają te trzy obiekty i ich prototyp. Zamiast współużytkować metody `toString` i `checkPassword` za pomocą prototypu, tu każdy egzemplarz zawiera kopię obu metod. W sumie jest więc sześć obiektów reprezentujących funkcje.

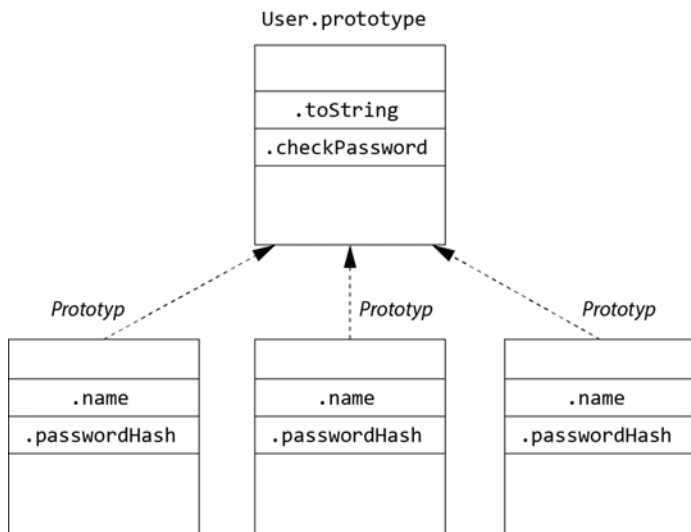


**Rysunek 4.3.** Przechowywanie metod w egzemplarzach

Rysunek 4.4 przedstawia te trzy obiekty i ich prototyp w pierwotnym rozwiązaniu. Metody `toString` i `checkPassword` są tworzone raz i współużytkowane przez wszystkie egzemplarze za pomocą prototypu.

Przechowywanie metod w prototypie sprawia, że są one dostępne we wszystkich egzemplarzach. Nie wymaga to tworzenia w każdym egzemplarzu wielu kopii funkcji lub dodatkowych właściwości. Możesz podejrzewać, że przechowywanie metod w egzemplarzach pozwala zoptymalizować szybkość znajdowania metod (takich jak `u3.toString()`), ponieważ nie jest konieczne wyszukiwanie implementacji metody `toString` w łańcuchu prototypów. Jednak w nowych silnikach JavaScriptu operacja wyszukiwania prototypu jest dobrze zoptymalizowana, dlatego kopiowanie metod do egzemplarzy nie gwarantuje zauważalnego wzrostu wydajności. Ponadto metody w egzemplarzach prawie zawsze zajmują więcej pamięci niż metody przechowywane w prototypie.





**Rysunek 4.4.** Przechowywanie metod w prototypie

### Co warto zapamiętać?

- Przechowywanie metod w egzemplarzach powoduje utworzenie wielu kopii funkcji (po jednej na egzemplarz).
- Przedkładaj przechowywanie metod w prototypach nad umieszczanie ich w egzemplarzach.

## Sposób 35. Stosuj domknięcia do przechowywania prywatnych danych

System obiektowy w JavaScriptcie nie zachęca do ukrywania informacji ani nie wymusza takiego zachowania. Nazwą każdej właściwości jest łańcuch znaków, a dowolna część programu ma dostęp do wszystkich właściwości obiektu — wystarczy podać odpowiednią nazwę. Pętle `for...in` oraz funkcje `Object.keys()` i `Object.getOwnPropertyNames()` ze standardu ES5 dodatkowo ułatwiają określenie nazw wszystkich właściwości obiektu.

Programiści JavaScriptu często uciekają się do stosowania konwencji, zamiast tworzyć w pełni prywatne właściwości. Na przykład niektóre osoby posługują się konwencjami dotyczącymi nazw właściwości i dodają przed nimi (lub po nich) podkreślenie (`_`). Nie ukrywa to właściwości, jednak sugeruje prawowładnym użytkownikom, że nie powinni sprawdzać ani modyfikować danej właściwości. Dzięki temu nie trzeba zmieniać implementacji obiektu.

Jednak w niektórych programach należy naprawdę ukryć właściwości. Załóżmy, że wymagająca zabezpieczeń platforma ma wysyłać obiekty do niezaufanej

aplikacji i programista chce uniemożliwić tej aplikacji modyfikowanie wewnętrznych mechanizmów tych obiektów. Innym miejscem, w którym wymuszanie ukrywania informacji jest przydatne, są często używane biblioteki. Tu mogą pojawić się trudne do wykrycia błędy, gdy nieostrożni użytkownicy przypadkowo będą polegać na szczegółach implementacji lub w nie ingerować.

Na potrzeby takich sytuacji JavaScript udostępnia dobry mechanizm ukrywania informacji. Są nim domknięcia.

Domknięcia to zamknięte struktury danych. Przechowują dane w wewnętrznych zmiennych i nie zapewniają bezpośredniego dostępu do nich. Dostęp do elementów domknięcia jest możliwy tylko wtedy, gdy funkcja jawnie go zapewnia. Oznacza to, że obiekty i domknięcia działają zgodnie z przeciwnymi zasadami. Właściwości obiektu są automatycznie dostępne, a zmienne domknięć są automatycznie ukryte.

Można wykorzystać tę cechę do umieszczenia w obiekcie w pełni prywatnych danych. Zamiast przechowywać dane jako właściwości obiektu, można zapisać je jako zmienne w konstruktorze i przekształcić metody obiektu w domknięcia używające tych zmiennych. Wróćmy do klasy `User` ze sposobu 30.

```
function User(name, passwordHash) {  
    this.toString = function() {  
        return "[User " + name + "]";  
    };  
    this.checkPassword = function(password) {  
        return hash(password) === passwordHash;  
    };  
}
```

Zauważ, że — w odróżnieniu od innych implementacji — metody `toString` i `checkPassword` używają `name` i `passwordHash` jak zmiennych, a nie jak właściwości obiektu `this`. Obecnie egzemplarz typu `User` nie zawiera żadnych właściwości, dlatego zewnętrzny kod nie ma bezpośredniego dostępu do nazwy i skrótu hasła z egzemplarza.

Wadą tego rozwiązania jest to, że aby zmienne z konstruktora mogły znaleźć się w zasięgu używających je metod, metody trzeba umieścić w egzemplarzu. Ze sposobu 34. dowiedziałeś się, że może to prowadzić do powstania dużej liczby kopii metod. Jednak w sytuacjach, gdy gwarancja ukrycia informacji jest ważna, warto ponieść ten koszt.

## Co warto zapamiętać?

- Zmienne w domknięciach są prywatne i dostępne tylko lokalnie.
- Stosuj zmienne lokalne jako prywatne dane, aby wymusić ukrywanie informacji w metodach.

## Sposób 36. Stan egzemplarzy przechowuj tylko w nich samych

Trzeba zrozumieć relację jeden do wielu między prototypem i jego egzemplarzami, aby móc zaimplementować poprawnie działające obiekty. Jednym z możliwych błędów jest przypadkowe zapisanie danych egzemplarza w prototypie. Na przykład w klasie reprezentującej drzewo może znajdować się tablica dzieci każdego węzła. Umieszczenie tej tablicy w prototypie prowadzi do zupełnie niepoprawnej implementacji.

```
function Tree(x) {
    this.value = x;
}

Tree.prototype = {
    children: [], // To powinno być stanem egzemplarza
    addChild: function(x) {
        this.children.push(x);
    }
};
```

Pomyśl, co się stanie, gdy spróbujesz zbudować drzewo za pomocą tej klasy.

```
var left = new Tree(2);
left.addChild(1);
left.addChild(3);

var right = new Tree(6);
right.addChild(5);
right.addChild(7);

var top = new Tree(4);
top.addChild(left);
top.addChild(right);

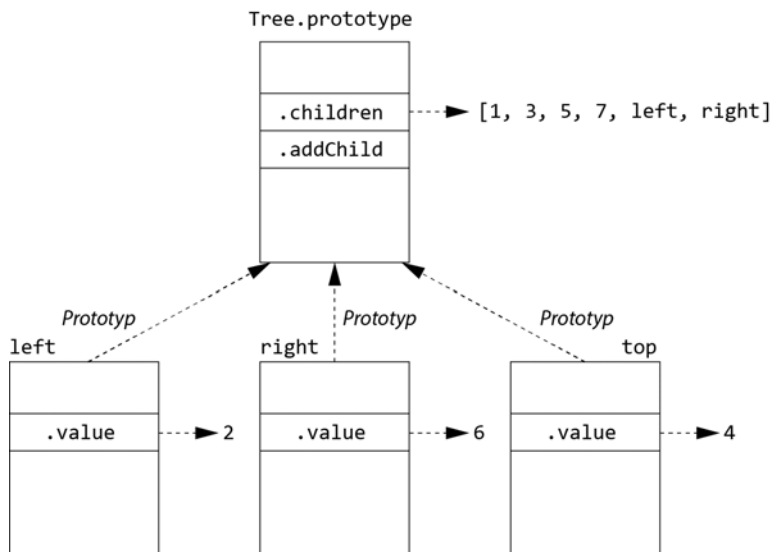
top.children; // [1, 3, 5, 7, left, right]
```

Przy każdym wywołaniu funkcji `addChild` do tablicy `Tree.prototype.children` dodawana jest wartość. W tablicy zapisywane są węzły zgodnie z kolejnością *wszystkich* wywołań funkcji `addChild`! Dlatego stan obiektów typu `Tree` jest niespójny, co przedstawia rysunek 4.5.

Poprawna implementacja klasy `Tree` polega na utworzeniu odrębnej tablicy `children` w każdym egzemplarzu.

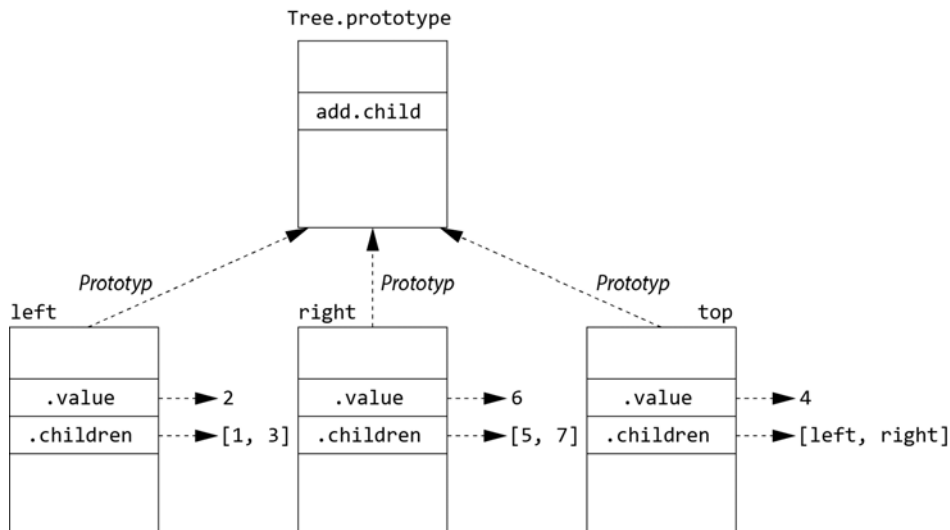
```
function Tree(x) {
    this.value = x;
    this.children = []; // Stan egzemplarza
}

Tree.prototype = {
    addChild: function(x) {
        this.children.push(x);
    }
};
```



**Rysunek 4.5.** Przechowywanie stanu egzemplarza w prototypie

Jeśli teraz uruchomisz pokazany wcześniej przykładowy kod, otrzymasz oczekiwany stan (przedstawiony na rysunku 4.6).



**Rysunek 4.6.** Przechowywanie stanu egzemplarza w egzemplarzach

Wniosek z tego jest taki, że dane reprezentujące stan mogą sprawiać problemy, gdy są współużytkowane. Metody zwykle można bezpiecznie współużytkować w wielu egzemplarzach klasy, ponieważ przeważnie są bezstanowe i jedynie używają stanu egzemplarza za pomocą referencji do obiektu `this`.

Ponieważ składnia wywołań gwarantuje, że nazwa `this` jest związana z egzemplarzem także w metodach odziedziczonych po prototypie, współużytkowane metody zachowują dostęp do stanu egzemplarza. Przeważnie bezpieczne jest współużytkowanie za pomocą prototypu również niezmiennych danych. Także dane reprezentujące stan można przechowywać w prototypie, o ile rzeczywiście powinny być współużytkowane. Jednak w prototypach najczęściej umieszcza się metody. Stan egzemplarza trzeba natomiast przechowywać w egzemplarzach.

### Co warto zapamiętać?

- Zmienne dane sprawiają problemy, gdy są współużytkowane. Prototypy są współużytkowane między wszystkimi ich egzemplarzami.
- Zmienny stan egzemplarza przechowuj w egzemplarzach.

## Sposób 37. Zwracaj uwagę na niejawne wiązanie obiektu `this`

Format CSV (ang. *comma-separated values*) to prosta tekstowa reprezentacja danych tabelarycznych.

Bösendorfer, 1828, Vienna, Austria

Fazioli, 1981, Sacile, Italy

Steinway, 1853, New York, USA

Można napisać prostą, modyfikowalną klasę do wczytywania danych w formacie CSV. Dla uproszczenia pomijamy tu możliwość parsowania danych ujętych w cudzysłów, takich jak "witaj, świecie". Format CSV ma różne postacie, w których funkcję separatorów pełnią inne znaki. Dlatego konstruktor przyjmuje opcjonalną tablicę separatorów i tworzy niestandardowe wyrażenie regularne używane do podziału wierszy na elementy.

```
function CSVReader(separators) {  
  this.separators = separators || [","];  
  this.regexp =  
    new RegExp(this.separators.map(function(sep) {  
      return "\\\""+ sep[0];  
    }).join("\\|"));  
}
```

Prosta implementacja metody `read` wykonuje dwa kroki. W pierwszym dzieli wejściowy łańcuch znaków na tablicę wierszy. W drugim rozбивa każdy wiersz na tablicę pojedynczych komórek. W efekcie powinna powstać dwuwymiarowa tablica łańcuchów znaków. Do wykonania tego zadania doskonale nadaje się metoda `map`.

```
CSVReader.prototype.read = function(str) {  
  var lines = str.trim().split(/\n/);  
  return lines.map(function(line) {
```

```

        return line.split(this.regex); // Niewłaściwe this
    });
};

```

```

var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n"); // ["a,b,c"], ["d,e,f"]

```

W tym na pozór prostym kodzie kryje się poważny, choć trudny do zauważenia błąd. Wywołanie zwrotne przekazywane do metody `lines.map` używa nazwy `this`, co ma służyć pobraniu właściwości `regex` obiektu `CSVReader`. Jednak metoda `map` ustawia jako odbiorcę wywołania zwrotnego tablicę `lines`, która nie ma wspomnianej właściwości. W efekcie wywołanie `this.regex` zwraca wartość `undefined`, a wywołanie `line.split` nie ma sensu.

Błąd wynika z tego, że nazwa `this` jest wiązana inaczej niż zmienne. W sposobach 18. i 25. wyjaśniono, że każda funkcja niejawnie wiąże nazwę `this` na podstawie miejsca wywołania funkcji. W przypadku zmiennych o zasięgu określonym leksykalnie zawsze można określić, gdzie są wiązane. Wystarczy sprawdzić jawne *wystąpienie wiązania* dla danej nazwy — na przykład na liście deklaracji `var` lub wśród parametrów funkcji. Natomiast nazwa `this` jest wiązana niejawnie przez najbliższą zewnętrzną funkcję. Dlatego wiązanie nazwy `this` w metodzie `CSVReader.prototype.read` różni się od wiązania tej nazwy w wywoływanej zwrotnie funkcji przekazywanej do metody `lines.map`.

Na szczęście, podobnie jak w przykładzie z wykorzystaniem metody `forEach` ze sposobu 25., można wykorzystać to, że metoda `map` tablic przyjmuje opcjonalny drugi argument wiązany z nazwą `this` w wywołaniu zwrotnym. Dlatego tu najłatwiejszym rozwiązaniem jest przekazanie zewnętrznego wiązania nazwy `this` do wywołania zwrotnego za pomocą drugiego argumentu metody `map`.

```

CSVReader.prototype.read = function(str) {
    var lines = str.trim().split(/\n/);
    return lines.map(function(line) {
        return line.split(this.regex);
    }, this); // Przekazywanie zewnętrznego wiązania nazwy this do wywołania zwrotnego
};

var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n");
// ["a", "b", "c"], ["d", "e", "f"]

```

Jednak nie wszystkie interfejsy API oparte na wywołaniach zwrotnych dają taką możliwość. Co można by zrobić, gdyby metoda `map` nie przyjmowała dodatkowego argumentu? Potrzebny byłby inny sposób na dostęp do wiązania nazwy `this` z zewnętrznej metody, tak aby wywołanie zwrotne mogło z niego skorzystać. Rozwiązanie jest proste. Wystarczy użyć zmiennej o zasięgu określonym leksykalnie i zapisać w niej dodatkową referencję do zewnętrznego wiązania nazwy `this`.

```

CSVReader.prototype.read = function(str) {
  var lines = str.trim().split(/\n/);
  var self = this; // Zapisanie referencji do zewnętrznego wiązania nazwy this
  return lines.map(function(line) {
    return line.split(self.regex); // Wykorzystanie zewnętrznego wiązania nazwy this
  });
};

var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n");
// [["a","b","c"],["d","e","f"]]

```

Programiści często stosują w tym wzorcu zmienną o nazwie `self`, co informuje, że zmienna służy wyłącznie jako alias wiązania nazwy `this` z bieżącego zasięgu. Inne często stosowane nazwy zmiennych w tym wzorcu to `me` i `that`. Używana nazwa nie ma tu większego znaczenia, jednak powszechnie stosowane nazwy ułatwiają innym programistom szybkie rozpoznanie wzorca.

Jeszcze innym podejściem, zgodnym ze standardem ES5, jest użycie metody `bind` wywoływanej zwrótnie funkcji. Podobne rozwiązanie opisano w sposobie 25.

```

CSVReader.prototype.read = function(str) {
  var lines = str.trim().split(/\n/);
  return lines.map(function(line) {
    return line.split(this.regex);
  }).bind(this); // Użycie zewnętrznego wiązania nazwy this
};

var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n");
// [["a","b","c"],["d","e","f"]]

```

### Co warto zapamiętać?

- Zasięg nazwy `this` jest zawsze wyznaczany przez najbliższą zawierającą ją funkcję.
- Stosuj zmienną lokalną (zwykle nazywaną `self`, `me` lub `that`) do udostępniania wiązania nazwy `this` w funkcjach wewnętrznych.

## Sposób 38. Wywoływanie konstruktorów klasy bazowej w konstruktorach klas pochodnych

**Graf sceny** to kolekcja obiektów reprezentujących scenę w programach graficznych (na przykład w grach lub symulacjach). Prosta scena obejmuje kolekcję wszystkich znajdujących się na niej obiektów (nazywanych **aktorami**), tablicę wstępnie wczytywanych danych graficznych powiązanych z aktorami i referencję do wyświetlacza (często nazywanego **kontekstem**).

```

function Scene(context, width, height, images) {
    this.context = context;
    this.width = width;
    this.height = height;
    this.images = images;
    this.actors = [];
}

Scene.prototype.register = function(actor) {
    this.actors.push(actor);
};

Scene.prototype.unregister = function(actor) {
    var i = this.actors.indexOf(actor);
    if (i >= 0) {
        this.actors.splice(i,1);
    }
};

Scene.prototype.draw = function() {
    this.context.clearRect(0, 0, this.width, this.height);
    for (var a = this.actors, i = 0, n = a.length;
        i < n;
        i++) {
        a[i].draw();
    }
};

```

Każdy z aktorów ze sceny dziedziczy po klasie bazowej `Actor`, obejmującej wspólne metody. Każdy aktor przechowuje referencję do sceny i swoje współrzędne oraz dodaje sam siebie do rejestru aktorów sceny.

```

function Actor(scene, x, y) {
    this.scene = scene;
    this.x = x;
    this.y = y;
    scene.register(this);
}

```

Aby umożliwić zmianę pozycji aktora na scenie, należy udostępnić metodę `moveTo`. Zmienia ona współrzędne aktora, a następnie ponownie wyświetla scenę.

```

Actor.prototype.moveTo = function(x, y) {
    this.x = x;
    this.y = y;
    this.scene.draw();
};

```

Gdy aktor opuszcza scenę, należy usunąć go z rejestru grafu sceny i ponownie ją wyświetlić.

```

Actor.prototype.exit = function() {
    this.scene.unregister(this);
    this.scene.draw();
};

```



W celu wyświetlenia aktora należy znaleźć jego obraz w tablicy danych graficznych grafu sceny. Zakładamy, że każdy aktor ma pole `type` pozwalające znaleźć jego obraz w tej tablicy. Po wczytaniu danych graficznych można wyświetlić je w kontekście za pomocą biblioteki graficznej. W tym przykładzie używamy interfejsu API w postaci HTML-owego elementu `Canvas`. Ten interfejs udostępnia metodę `drawImage` służącą do wyświetlania obiektów typu `Image` w elemencie `<canvas>` strony internetowej.

```
Actor.prototype.draw = function() {  
    var image = this.scene.images[this.type];  
    this.scene.context.drawImage(image, this.x, this.y);  
};
```

Na podstawie danych graficznych można też ustalić wymiary aktora.

```
Actor.prototype.width = function() {  
    return this.scene.images[this.type].width;  
};  
  
Actor.prototype.height = function() {  
    return this.scene.images[this.type].height;  
};
```

Konkretne rodzaje aktorów są zaimplementowane jako klasy pochodne od klasy `Actor`. Na przykład statek kosmiczny w grze zręcznościowej jest reprezentowany przez klasę `SpaceShip` pochodną od klasy `Actor`. `SpaceShip`, podobnie jak wszystkie klasy, jest zdefiniowana za pomocą konstruktora. Jednak aby mieć pewność, że egzemplarze tej klasy zostaną poprawnie zainicjowane jako aktorzy, w jej konstruktorze trzeba jawnie wywoływać konstruktor klasy `Actor`. W tym celu należy wywołać konstruktor klasy `Actor` z nowym obiektem ustawionym jako odbiorcą.

```
function SpaceShip(scene, x, y) {  
    Actor.call(this, scene, x, y);  
    this.points = 0;  
}
```

Wywołanie najpierw konstruktora klasy `Actor` gwarantuje, że do nowego obiektu zostaną dodane wszystkie właściwości egzemplarza tworzone za pomocą tego konstruktora. Następnie w konstruktorze `SpaceShip` można zdefiniować właściwości egzemplarzy tej klasy pochodnej, na przykład liczbę punktów zdobytych przez statek.

Aby `SpaceShip` była klasą pochodną klasy `Actor`, jej prototyp musi dziedziczyć po prototypie `Actor.prototype`. Do uzyskania potrzebnego efektu najlepiej jest wykorzystać metodę `Object.create` ze standardu ES5.

```
SpaceShip.prototype = Object.create(Actor.prototype);
```

W sposobie 33. opisano, jak dodać metodę `Object.create` w środowiskach, które nie obsługują standardu ES5. Próba utworzenia prototypu klasy `SpaceShip` za

pomocą konstruktora klasy `Actor` związana jest z kilkoma problemami. Pierwszy z nich dotyczy braku sensownych argumentów dla konstruktora `Actor`.

```
SpaceShip.prototype = newActor();
```

W momencie inicjowania prototypu klasy `SpaceShip` nie są jeszcze gotowe sceny, które można by przekazać jako pierwszy argument. Ponadto prototyp klasy `SpaceShip` nie ma współrzędnych  $x$  i  $y$ . Te właściwości powinny należeć do poszczególnych egzemplarzy typu `SpaceShip`, a nie do prototypu `SpaceShip.prototype`. Jeszcze większym problemem jest to, że konstruktor klasy `Actor` dodaje obiekt do rejestru sceny, czego nie należy robić z prototypem klasy `SpaceShip`. Jest to częsta sytuacja w klasach pochodnych — konstruktor klasy bazowej należy wywoływać tylko w konstruktorze klasy pochodnej, a nie przy tworzeniu prototypu tej ostatniej.

Po utworzeniu prototypu klasy `SpaceShip` można dodać do niego wszystkie właściwości współużytkowane w egzemplarzach, w tym nazwę typu służącą do wskazywania elementów w tablicy danych graficznych sceny i metody specyficzne dla statków kosmicznych.

```
SpaceShip.prototype.type = "spaceShip";
```

```
SpaceShip.prototype.scorePoint = function() {  
    this.points++;  
};
```

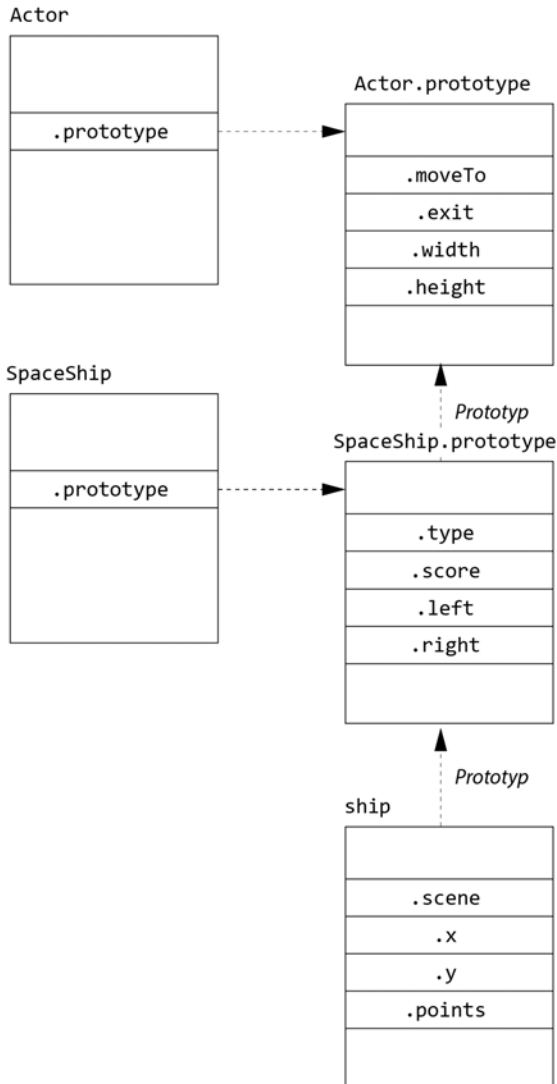
```
SpaceShip.prototype.left = function() {  
    this.moveTo(Math.max(this.x - 10, 0), this.y);  
};
```

```
SpaceShip.prototype.right = function() {  
    var maxWidth = this.scene.width - this.width();  
    this.moveTo(Math.min(this.x + 10, maxWidth), this.y);  
};
```

Rysunek 4.7 przedstawia hierarchię dziedziczenia egzemplarzy typu `SpaceShip`. Zauważ, że właściwości `scene`,  $x$  i  $y$  są zdefiniowane tylko dla egzemplarzy, a nie dla prototypu, choć są tworzone przez konstruktor klasy `Actor`.

### Co warto zapamiętać?

- Jawnie wywołuj konstruktor klasy bazowej w konstruktorze klasy pochodnej. Bezpośrednio ustawiaj przy tym obiekt `this` jako odbiorcę.
- Za pomocą metody `Object.create` twórz prototyp klasy pochodnej, aby uniknąć wywoływania konstruktora klasy bazowej.



**Rysunek 4.7.** Hierarchia dziedziczenia z klasami pochodnymi

### Sposób 39. Nigdy nie wykorzystuj ponownie nazw właściwości z klasy bazowej

Załóżmy, że programista chce dodać do przedstawionej w sposobie 38. biblioteki grafu sceny mechanizm zbierania danych diagnostycznych, który może okazać się przydatny do debugowania lub profilowania kodu. W tym celu można do każdego egzemplarza typu **Actor** dodać unikatowy numer identyfikacyjny.

```
function Actor(scene, x, y) {
  this.scene = scene;
  this.x = x;
  this.y = y;
  this.id = ++Actor.nextID;
  scene.register(this);
}
```

```
Actor.nextID = 0;
```

Następnie można zrobić to samo dla poszczególnych egzemplarz klasy pochodnej od klasy Actor. Niech będzie to klasa Alien reprezentująca wrogów statku kosmicznego. Ta klasa ma obok numeru identyfikacyjnego aktora zawierać odrębny numer identyfikacyjny kosmity.

```
function Alien(scene, x, y, direction, speed, strength) {
  Actor.call(this, scene, x, y);
  this.direction = direction;
  this.speed = speed;
  this.strength = strength;
  this.damage = 0;
  this.id = ++Alien.nextID; // Powoduje konflikt z właściwością id klasy Actor
}
```

```
Alien.nextID = 0;
```

Ten kod powoduje konflikt między klasą Alien a klasą bazową Actor. Obie klasy próbują zapisać wartość właściwości id egzemplarza. Choć w każdej klasie ta właściwość jest „prywatna” (czyli ważna i dostępna tylko dla metod zdefiniowanych bezpośrednio w danej klasie), jest przechowywana w egzemplarzach i ma nazwę w postaci łańcucha znaków. Jeśli dwie klasy w hierarchii dziedziczenia używają identycznej nazwy właściwości, dotyczy ona tej samej właściwości.

Dlatego w klasach pochodnych zawsze trzeba uwzględniać wszystkie właściwości używane w klasach bazowych — nawet jeśli te właściwości są uznawane za prywatne. Oczywistym rozwiązaniem opisanego problemu jest zastosowanie różnych nazw właściwości dla numerów identyfikacyjnych w klasach Actor i Alien.

```
function Actor(scene, x, y) {
  this.scene = scene;
  this.x = x;
  this.y = y;
  this.actorID = ++Actor.nextID; // Różne od alienID
  scene.register(this);
}
```

```
Actor.nextID = 0;
```

```
function Alien(scene, x, y, direction, speed, strength) {
  Actor.call(this, scene, x, y);
  this.direction = direction;
  this.speed = speed;
```

```

    this.strength = strength;
    this.damage = 0;
    this.alienID = ++Alien.nextID; // Różne od actorID
}

Alien.nextID = 0;

```

### Co warto zapamiętać?

- Uwzględniaj wszystkie nazwy właściwości używane w klasach bazowych.
- Nigdy nie stosuj nazw właściwości z klasy bazowej w klasach pochodnych.

## Sposób 40. Unikaj dziedziczenia po klasach standardowych

Biblioteka standardowa języka ECMAScript jest niewielka, ale udostępnia zestaw ważnych klas, takich jak `Array`, `Function` i `Date`. Rozszerzanie tych klas może wydawać się dobrym pomysłem, ale ich definicje niestety zawierają specjalne operacje, powodujące, że napisanie poprawnie działających klas pochodnych jest niemożliwe.

Dobrym przykładem jest klasa `Array`. W bibliotece do manipulowania systemem plików przydatna może być reprezentująca katalogi klasa dziedzicząca wszystkie mechanizmy tablic.

```

function Dir(path, entries) {
    this.path = path;
    for (var i = 0, n = entries.length; i < n; i++) {
        this[i] = entries[i];
    }
}

```

```

Dir.prototype = Object.create(Array.prototype);
// Rozszerzanie klasy Array

```

Niestety, w tym rozwiązaniu właściwość `length` tablic działa nieprawidłowo.

```

var dir = new Dir("/tmp/mysite",
                  ["index.html", "script.js", "style.css"]);
dir.length; // 0

```

Dzieje się tak, ponieważ właściwość `length` operuje na obiektach oznaczonych wewnętrznie jako „prawdziwe” tablice. Standard ECMAScript określa je na podstawie ukrytej **właściwości wewnętrznej** o nazwie `[[Class]]`. Niech ta nazwa Cię nie zmyli. JavaScript nie używa wewnętrznego systemu klas. Wartość właściwości `[[Class]]` to zwykła nazwa. Dla tablic (utworzonych za pomocą konstruktora `Array` lub składni `[]`) wartość właściwości `[[Class]]` to `"Array"`, dla funkcji ta wartość to `"Function"` itd. W tabeli 4.1 znajdziesz kompletny zestaw wartości właściwości `[[Class]]` zdefiniowanych w standardzie ECMAScript.

**Tabela 4.1.** Wartości wewnętrznej właściwości `[[Class]]` zdefiniowane w standardzie ECMAScript

<code>[[Class]]</code>	Instrukcje
"Array"	<code>new Array(...), [...]</code>
"Boolean"	<code>new Boolean(...)</code>
"Date"	<code>new Date(...)</code>
"Error"	<code>new Error(...), new EvalError(...), new RangeError(...), new ReferenceError(...), new SyntaxError(...), new TypeError(...), new URIError(...)</code>
"Function"	<code>new Function(...), function(...) {...}</code>
"JSON"	JSON
"Math"	Math
"Number"	<code>new Number(...)</code>
"Object"	<code>new Object(...), {...}, new MyClass(...)</code>
"RegExp"	<code>new RegExp(...), /.../</code>
"String"	<code>new String(...)</code>

Co owa magiczna właściwość `[[Class]]` ma wspólnego z właściwością `length`? Okazuje się, że działanie właściwości `length` jest zdefiniowane konkretnie dla obiektów, których wewnętrzna właściwość `[[Class]]` ma wartość "Array". Dla tych obiektów właściwość `length` jest synchronizowana z liczbą indeksowanych właściwości danego obiektu. Gdy programista doda indeksowane właściwości, właściwość `length` zostanie automatycznie powiększona. Zmniejszenie wartości właściwości `length` prowadzi do automatycznego usunięcia indeksowanych właściwości wykraczających poza nową wartość.

Jednak po rozszerzeniu klasy `Array` egzemplarze klas pochodnych nie są tworzone za pomocą instrukcji `new Array()` ani składni `[]`. Dlatego dla egzemplarzy typu `Dir` wartość właściwości `[[Class]]` to "Object". Można się o tym przekonać. Domyślna metoda `Object.prototype.toString` pobiera wartość wewnętrznej właściwości `[[Class]]` odbiorcy, aby utworzyć opis obiektu. Można więc jawnie wywołać tę metodę dla dowolnego obiektu i sprawdzić zwróconą wartość.

```
var dir = new Dir("/", []);
Object.prototype.toString.call(dir); // "[object Object]"
Object.prototype.toString.call([]); // "[object Array]"
```

Tak więc egzemplarze typu `Dir` nie dziedziczą oczekiwanego specjalnego działania właściwości `length` tablic.

Oto lepsza implementacja, w której zdefiniowana jest właściwość egzemplarza z tablicą elementów.

```
function Dir(path, entries) {
  this.path = path;
```

```
    this.entries = entries; // Właściwość reprezentująca tablicę  
}
```

Metody tablicy można zredefiniować w prototypie. W tym celu należy delegować zadania do odpowiednich metod właściwości `entries`.

```
Dir.prototype.forEach = function(f, thisArg) {  
    if (typeof thisArg === "undefined") {  
        thisArg = this;  
    }  
    this.entries.forEach(f, thisArg);  
};
```

Większość konstruktorów z biblioteki standardowej ECMAScript powoduje podobne problemy. Niektóre właściwości lub metody wymagają odpowiedniej wartości właściwości `[[Class]]` lub innych specjalnych właściwości wewnętrznych, których nie da się udostępnić w klasach pochodnych. Dlatego zaleca się unikanie dziedziczenia po następujących klasach standardowych: `Array`, `Boolean`, `Date`, `Function`, `Number`, `RegExp` i `String`.

### Co warto zapamiętać?

- Dziedziczenie po klasach standardowych zwykle kończy się niepowodzeniem. Przyczyną są specjalne właściwości wewnętrzne, na przykład `[[Class]]`.
- Przedkładaj delegowanie zadań do właściwości nad dziedziczenie po klasach standardowych.

## Sposób 41. Traktuj prototypy jak szczegół implementacji

Obiekt udostępnia klientom niewielki, prosty i wartościowy zestaw operacji. Najbardziej podstawowe formy interakcji klienta z obiektem polegają na pobieraniu wartości właściwości i wywoływaniu metod. W tych operacjach nie ma znaczenia, gdzie w hierarchii prototypu przechowywane są te właściwości. Implementacja obiektu nieraz z czasem się zmienia, co może prowadzić do przeniesienia właściwości w inne miejsce w łańcuchu prototypów obiektu. Jednak dopóki wartość właściwości będzie taka sama, podstawowe operacje będą przebiegać w identyczny sposób. Można ująć to prosto — w kontekście działania obiektu prototyp to szczegół implementacji.

JavaScript udostępnia wygodne mechanizmy **introspekcji** umożliwiające inspekcję szczegółów obiektów. Metoda `Object.prototype.hasOwnProperty` określa, czy właściwość jest przechowywana bezpośrednio jako „własna” w obiekcie (jest wtedy właściwością egzemplarza), co pozwala zignorować hierarchię prototypów. Mechanizmy `Object.getPrototypeOf` i `__proto__` (zobacz Sposób 30.) umożliwiają programom poruszanie się po łańcuchu prototypów obiektu

i przeszukiwanie poszczególnych prototypów. Te mechanizmy dają duże możliwości i czasem okazują się przydatne.

Jednak dobry programista wie, kiedy należy respektować granice abstrakcyjnych jednostek. Inspekcja szczegółów implementacji (nawet bez ich modyfikowania) prowadzi do zależności między komponentami programu. Jeśli autor obiektu zmieni szczegóły implementacji, zależny od nich klient przestanie działać. Diagnoza błędów tego rodzaju jest trudna, ponieważ wymaga **pracy na odległość** — jeden programista zmienia implementację jednego komponentu, a inny komponent (często napisany przez kogoś innego) przestaje działać.

JavaScript nie odróżnia publicznych właściwości obiektu od prywatnych (zobacz Sposób 35.). Programista musi polegać na dokumentacji i dyscyplinie. Jeśli biblioteka udostępnia obiekt z właściwościami, które są nieudokumentowane lub opisane jako wewnętrzne, prawdopodobnie klienci nie powinny z nich korzystać.

### Co warto zapamiętać?

- Obiekty to interfejsy, a prototypy to implementacje.
- Unikaj inspekcji struktury prototypów obiektów, których nie kontrolujesz.
- Unikaj inspekcji właściwości z wewnętrznej implementacji obiektów, których nie kontrolujesz.

## Sposób 42. Unikaj nieprzemyślanego stosowania techniki monkey patching

Po zaprezentowanej w sposobie 41. krytyce naruszania granic abstrakcyjnych jednostek przejdźmy do skrajnego przypadku takiej sytuacji. Ponieważ prototypy są współużytkowane jako obiekty, każdy może dodawać, usuwać i modyfikować ich właściwości. Ta kontrowersyjna technika nosi nazwę *monkey patching*.

Atrakcyjność tej techniki wynika z możliwości, jakie oferuje. Czy w tablicach brakuje przydatnej metody? Można ją dodać samodzielnie.

```
Array.prototype.split = function(i) { // Wersja numer 1
    return [this.slice(0, i), this.slice(i)];
};
```

*Voilà* — od teraz każdy egzemplarz tablicy ma metodę `split`.

Problemy pojawiają się, gdy w różnych bibliotekach te same prototypy zostaną wzbogacone w niezgodny sposób. Autor innej biblioteki może zmodyfikować prototyp `Array.prototype` przez dodanie metody o użytej wcześniej nazwie.



```
Array.prototype.split = function() { // Wersja numer 2
    var i = Math.floor(this.length / 2);
    return [this.slice(0, i), this.slice(i)];
};
```

Teraz przy każdym wywołaniu metody `split` tablic występuje prawdopodobieństwo około 50%, że wynik będzie nieprawidłowy (zależy to od tego, której wersji oczekuje użytkownik).

Należy zadbać przynajmniej o to, aby w każdej bibliotece modyfikującej współużytkowane prototypy, takie jak `Array.prototype`, wyraźnie to udokumentować. Zapewni to użytkownikom ostrzeżenie o możliwych konfliktach między bibliotekami. Jednak dwóch bibliotek, które modyfikują prototypy w niezgodny sposób, nie można używać w tym samym programie. Jedno z rozwiązań polega na tym, że w bibliotekach modyfikujących prototypy dla wygody użytkowników można wprowadzać zmiany w funkcji. Użytkownik może wywołać tę funkcję lub ją zignorować.

```
function addArrayMethods() {
    Array.prototype.split = function(i) {
        return [this.slice(0, i), this.slice(i)];
    };
};
```

To podejście sprawdza się oczywiście tylko wtedy, gdy biblioteka udostępniająca funkcję `addArrayMethods` nie wymaga do działania funkcji `Array.prototype.split`.

Mimo zagrożeń istnieje stabilne i wartościowe zastosowanie techniki monkey patching. Mianowicie można ją wykorzystać do tworzenia **wypełniaczy** (ang. *polyfill*). Platformy i biblioteki JavaScriptu często są instalowane w różnych platformach, na przykład w rozmaitych wersjach przeglądarek różnych producentów. Te programy mogą różnić się ze względu na implementację wielu standardowych interfejsów API. Na przykład w standardzie ES5 zdefiniowano nowe metody typu `Array` (na przykład `forEach`, `map` i `filter`), które w niektórych przeglądarkach nie są jeszcze obsługiwane. Działanie brakujących metod jest zdefiniowane w powszechnie obsługiwanym standardzie, dlatego wiele programów i bibliotek może wymagać tych metod. Ponieważ ich działanie nie jest opisane w standardzie, implementowanie tych metod nie jest związane z opisanym wcześniej ryzykiem niezgodności między bibliotekami. Wiele bibliotek może udostępniać implementacje tych samych metod standardowych (przy założeniu, że wszystkie te implementacje są poprawne), ponieważ są one oparte na tym samym standardowym interfejsie API.

Dlatego w odpowiednich platformach można bezpiecznie wypełnić luki, dodając do techniki *monkey patching* test.

```
if (typeof Array.prototype.map !== "function") {
    Array.prototype.map = function(f, thisArg) {
        var result = [];
        for (var i = 0, n = this.length; i < n; i++) {
```

```
        result[i] = f.call(thisArg, this[i], i);
    }
    return result;
};
}
```

Sprawdzanie dostępności funkcji `Array.prototype.map` pozwala się upewnić, że wbudowana instrukcja (która zwykle jest wydajniejsza i dokładniej przetworzana) nie zostanie zastąpiona.

### Co warto zapamiętać?

- Unikaj nieprzemyślanego stosowania techniki *monkey patching*.
- Udokumentuj wszelkie przypadki zastosowania tej techniki w bibliotece.
- Rozważ umożliwienie opcjonalnego wprowadzania modyfikacji za pomocą wywołania eksportowanej funkcji.
- Stosuj technikę monkey patching do zapewniania wypełniaczy dla brakujących standardowych interfejsów API.

# 5

## Tablice i słowniki

Obiekty to najbardziej wszechstronna struktura danych w JavaScriptcie. W zależności od sytuacji mogą reprezentować rekord z parami nazwa – wartość, obiektową abstrakcję danych z odziedziczonymi metodami, gęstą lub rzadką tablicę, tablicę z haszowaniem itd. Oczywiście opanowanie takiego wielofunkcyjnego narzędzia wymaga poznania różnych idiomów odpowiadających poszczególnym potrzebom. Z tego rozdziału dowiesz się, jak używać ustrukturyzowanych obiektów i jak wykorzystać mechanizm dziedziczenia. Zobaczysz też, jak używać obiektów w postaci **kolekcji**, czyli zagregowanych struktur danych o zmiennej liczbie elementów.

### Sposób 43. Budowanie prostych słowników na podstawie egzemplarzy typu Object

Obiekt w JavaScriptcie to tablica łącząca nazwy (w postaci łańcuchów znaków) właściwości z ich wartościami. Dlatego obiekty są wygodnie prostym narzędziem do implementowania **słowników**, czyli kolekcji o zmiennej długości łączących łańcuchy znaków z wartościami. JavaScript udostępnia nawet wygodny mechanizm do enumeracji nazw właściwości obiektu. Jest nim pętla `for...in`.

```
var dict = { alicja: 34, robert: 24, maciej: 62 };  
var people = [];
```

```
for (var name in dict) {  
    people.push(name + ": " + dict[name]);  
}
```

```
people; // ["alicja: 34", "robert: 24", "maciej: 62"]
```

Jednak każdy obiekt dziedziczy także właściwości po prototypie (zobacz rozdział 4.), a pętla `for...in` enumeruje zarówno odziedziczone, jak i „własne” właściwości obiektu. Co się stanie, jeśli utworzysz niestandardową klasę słownika przechowującą elementy jako właściwości samego obiektu słownika?

```
function NaiveDict() { }

NaiveDict.prototype.count = function() {
  var i = 0;
  for (var name in this) { // Zlicza wszystkie właściwości
    i++;
  }
  return i;
};

NaiveDict.prototype.toString = function() {
  return "[Object NaiveDict]";
};

var dict = new NaiveDict();

dict.alicja = 34;
dict.robert = 24;
dict.maciej = 62;

dict.count(); // 5
```

Problem polega na tym, że ten sam obiekt jest używany do przechowywania zarówno stałych właściwości struktury danych `NaiveDict` (`count` i `toString`), jak i zmiennych elementów z określonego słownika (`alicja`, `robert`, `maciej`). Dlatego gdy funkcja `count` zlicza właściwości słownika, uwzględnia wszystkie te jednostki (`count`, `toString`, `alicja`, `robert`, `maciej`) zamiast tylko istotne tu elementy słownika. W sposobie 45. przedstawiona jest poprawiona klasa `Dict`, która nie przechowuje elementów jako właściwości egzemplarzy. Zamiast tego udostępnia metody `dict.get(key)` i `dict.set(key, value)`. Jednak niniejsze sposób dotyczy wzorca używania właściwości obiektu jako elementów słownika.

Podobnym błędem jest użycie do reprezentowania słowników typu `Array`. W tę pułapkę często wpadają zwłaszcza programiści używający Perla i PHP, gdzie słowniki nazywane są tablicami asocjacyjnymi. Ponieważ w JavaScriptcie można dodawać właściwości do dowolnych typów, to rozwiązanie *czasem* działa, co dodatkowo myli programistów.

```
var dict = new Array();

dict.alicja = 34;
dict.robert = 24;
dict.maciej = 62;

dict.robert; // 24
```

Niestety, ten kod jest podatny na problem **zaśmiecania przez prototypy** (ang. *prototype pollution*). Polega to na tym, że właściwości prototypu mogą spowodować pojawianie się nieoczekiwanych właściwości przy enumeracji elementów słownika. Na przykład autor innej biblioteki z aplikacji może zdecydować się dodać do prototypu `Array.prototype` metody pomocnicze.

```
Array.prototype.first = function() {  
    return this[0];  
};
```

```
Array.prototype.last = function() {  
    return this[this.length - 1];  
};
```

Co się stanie, gdy spróbujesz wyświetlić elementy z tablicy?

```
var names = [];  
  
for (var name in dict) {  
    names.push(name);  
}  
  
names; // ["alicja", "robert", "maciej", "first", "last"]
```

To prowadzi do podstawowej reguły stosowania obiektów jako prostych słowników. Przy tworzeniu słowników korzystaj tylko z egzemplarzy typu `Object` zamiast klas pochodnych (takich jak `NaiveDict`) lub tablic. W przedstawionym kodzie wystarczy zastąpić fragment `new Array()` instrukcją `new Object()` lub nawet pustym literałem obiekowym. Uzyskany kod będzie znacznie mniej podatny na zaśmiecanie przez prototypy.

```
var dict = {};  
  
dict.alicja = 34;  
dict.robert = 24;  
dict.maciej = 62;  
  
var names = [];  
  
for (var name in dict) {  
    names.push(name);  
}  
  
names; // ["alicja", "robert", "maciej"]
```

Nowa wersja nie jest w pełni zabezpieczona przed zaśmiecaniem. Każdy może dodać właściwości do prototypu `Object.prototype`, co znów doprowadzi do kłopotów. Jednak dzięki zastosowaniu typu `Object` wiadomo, że ryzyko związane jest tylko z prototypem `Object.prototype`.

Dlaczego nowe rozwiązanie jest lepsze? W sposobie 47. wyjaśniono, że nikt *nigdy* nie powinien dodawać do prototypu `Object.prototype` właściwości, które mogłyby zaśmiecać pętlę `for...in`. Natomiast dodawanie właściwości do

prototypu `Array.prototype` czasem jest uzasadnione. Na przykład w sposobie 42. opisano, jak dodawać standardowe metody do tego prototypu w środowiskach, które ich nie udostępniają. Te właściwości zaśmiecają pętlę `for...in`. Podobnie w prototypach klas zdefiniowanych przez użytkowników zwykle znajdują się właściwości. Ograniczenie się do stosowania egzemplarzy typu `Object` (i przestrzeganie reguły ze sposobu 47.) pozwala uniknąć zaśmiecania pętli `for...in`.

Uważaj jednak! Ze sposobów 44. i 45. dowiesz się, że wspomniana reguła jest konieczna, ale niewystarczająca do uzyskania poprawnie działającego słownika. Choć proste słowniki są wygodne w użyciu, są też narażone na wiele problemów. Dlatego powinieneś zapoznać się z wszystkimi trzema sposobami, a jeśli nie chcesz zapamiętywać reguł, zastosować abstrakcję taką jak klasa `Dict` ze sposobu 45.

### Co warto zapamiętać?

- Do tworzenia prostych słowników stosuj literały obiektowe.
- Proste słowniki powinny być oparte bezpośrednio na prototypie `Object.prototype`, co chroni przed zaśmiecaniem pętli `for...in` przez prototypy.

### Sposób 44. Stosuj prototypy `null`, aby uniknąć zaśmiecania przez prototypy

Jednym z najłatwiejszych sposobów na uniknięcie zaśmiecania przez prototypy jest uniemożliwienie `go`. Przed wprowadzeniem specyfikacji ES5 nie istniał standardowy sposób tworzenia nowych obiektów z pustym prototypem. Możesz spróbować ustawić właściwość `prototype` konstruktora na `null` lub `undefined`.

```
function C() { }  
C.prototype = null;
```

Jednak wywołanie tego konstruktora i tak spowoduje utworzenie egzemplarza typu `Object`.

```
var o = new C();  
Object.getPrototypeOf(o) === null;           // false  
Object.getPrototypeOf(o) === Object.prototype; // true
```

Specyfikacja ES5 zapewnia pierwszy standardowy sposób tworzenia obiektów bez prototypu. Funkcja `Object.create` potrafi dynamicznie tworzyć obiekty na podstawie podanego przez użytkownika prototypu i **mapy deskryptorów właściwości**, opisującej wartości i cechy właściwości nowego obiektu. Przekazanie `null` jako argumentu reprezentującego prototyp i pustej mapy deskryptorów pozwala utworzyć pusty obiekt.

```
var x = Object.create(null);
Object.getPrototypeOf(o) === null; // true
```

Zaśmiecanie przez prototypy nie wpływa na działanie takich obiektów.

W starszych środowiskach, nieobsługujących metody `Object.create`, można zastosować inne godne uwagi rozwiązanie. W wielu środowiska specjalna właściwość `__proto__` (zobacz sposoby 31. i 32.) zapewnia „magiczny” dostęp w trybie odczytu i zapisu do wewnętrznego wiązania obiektu z prototypem. Składnia literału obiektowego umożliwia zainicjowanie tego wiązania w nowym obiekcie wartością `null`.

```
var x = { __proto__: null };
x instanceof Object; // false (niestandardowe rozwiązanie)
```

Ta składnia jest wygodna, jeśli jednak metoda `Object.create` jest dostępna, stanowi bezpieczniejsze rozwiązanie. Właściwość `__proto__` jest niestandardowa i w niektórych platformach nie występuje. Nie ma gwarancji, że w przyszłych implementacjach JavaScriptu ta właściwość będzie dostępna, dlatego tam, gdzie to możliwe, należy ograniczyć się do używania metody `Object.create`.

Niestety, choć niestandardowa właściwość `__proto__` rozwiązuje pewne problemy, *powoduje* inne kłopoty i sprawia, że utworzone przy jej użyciu obiekty bez prototypów nie są niezawodną implementacją słowników. W sposobie 45. opisano, że w niektórych środowiskach JavaScriptu sam klucz właściwości ("`__proto__`") zaśmieca obiekty, *nawet gdy nie mają one prototypu*. Jeśli nie masz pewności, że łańcuch znaków "`__proto__`" nigdy nie będzie używany jako klucz w słowniku, pomyśl o zastosowaniu bardziej odpornej na problemy klasy `Dict` przedstawionej w sposobie 45.

## Co warto zapamiętać?

- W środowiskach zgodnych ze standardem ES5 używaj wywołania `Object.create(null)` do tworzenia pustych obiektów bez prototypów. Są one bardziej odporne na zaśmiecanie.
- W starszych środowiskach możesz zastosować instrukcję `{ __proto__: null }`.
- Pamiętaj jednak, że właściwość `__proto__` nie jest standardowa ani w pełni przenośna. W przyszłości może też zostać usunięta ze środowisk JavaScriptu.
- Nigdy nie stosuj nazwy "`__proto__`" jako klucza w słowniku, ponieważ niektóre środowiska traktują tę właściwość w specjalny sposób.

## Sposób 45. Używaj metody `hasOwnProperty` do zabezpieczania się przed zaśmiecaniem przez prototypy

W sposobach 43. i 44. opisano *enumerację* właściwości, jednak nie wyjaśniono problemu zaśmiecania przez prototypy w kontekście *wyszukiwania* właściwości. Na pozór dobrym rozwiązaniem jest wykorzystanie wbudowanej w JavaScript składni manipulowania obiektami do wszystkich operacji na słowniku.

```
"alicja" in dict:    // Test przynależności
dict.alicja:        // Pobieranie
dict.alicja = 24;    // Aktualizowanie
```

Pamiętaj jednak, że w JavaScriptcie w operacjach na obiektach zawsze uwzględniane jest dziedziczenie. Nawet pusty literal obiektowy dziedziczy liczne właściwości po prototypie `Object.prototype`.

```
var dict = {};

"alicja" in dict:    // false
"robert" in dict:    // false
"maciej" in dict:    // false
"toString" in dict:  // true
"valueOf" in dict:   // true
```

Na szczęście prototyp `Object.prototype` udostępnia metodę `hasOwnProperty`, która pozwala uniknąć zaśmiecania przez prototypy przy sprawdzaniu elementów słownika.

```
dict.hasOwnProperty("alicja");    // false
dict.hasOwnProperty("toString"); // false
dict.hasOwnProperty("valueOf");  // false
```

Przed zaśmiecaniem przez prototypy przy wyszukiwaniu właściwości można też się zabezpieczyć, dodając do wyszukiwania test.

```
dict.hasOwnProperty("alicja") ? dict.alicja : undefined;
dict.hasOwnProperty(x) ? dict[x] : undefined;
```

Niestety, to jeszcze nie wszystko. Wywołanie `dict.hasOwnProperty` wymaga znalezienia metody `hasOwnProperty` obiektu `dict`. Standardowo metoda ta jest dziedziczona po prototypie `Object.prototype`. Jeśli jednak zapiszesz w słowniku element o nazwie `"hasOwnProperty"`, metoda prototypu przestanie być dostępna.

```
dict.hasOwnProperty = 10;
dict.hasOwnProperty("alicja");
// Błąd: dict.hasOwnProperty nie jest funkcją
```

Może Ci się wydawać, że w słowniku nigdy nie znajdzie się element o nazwie tak nietypowej jak `"hasOwnProperty"`. Oczywiście możesz uznać, że w danym programie taki element nigdy się nie pojawi. Jednak może on wystąpić, zwłaszcza



jeśli elementy słownika są pobierane z zewnętrznego pliku, zasobu sieciowego lub od użytkownika. Nie masz wtedy kontroli nad tym, jakie klucze znajdują się w słowniku.

Najbezpieczniejszym podejściem jest unikanie założeń. Zamiast wywoływać `hasOwnProperty` jako metodę słownika, można wykorzystać metodę `call` opisaną w sposobie 20. Najpierw należy pobrać metodę `hasOwnProperty` z dobrze znanej lokalizacji.

```
var hasOwn = Object.prototype.hasOwnProperty;
```

Oto bardziej zwięzły zapis:

```
var hasOwn = {}.hasOwnProperty;
```

Teraz gdy zmienna lokalna jest już związana z odpowiednią funkcją, można wywołać ją dla dowolnego obiektu za pomocą metody `call` funkcji.

```
hasOwn.call(dict, "alicja");
```

To podejście działa niezależnie od tego, czy w odbiorcy przesłonięta jest metoda `hasOwnProperty`.

```
var dict = {};
```

```
dict.alicja = 24;
hasOwn.call(dict, "hasOwnProperty"); // false
hasOwn.call(dict, "alicja");          // true
```

```
dict.hasOwnProperty = 10;
hasOwn.call(dict, "hasOwnProperty"); // true
hasOwn.call(dict, "alicja");          // true
```

Aby uniknąć wstawiania tego szablonowego kodu przy każdym przeszukiwaniu, można zapisać przedstawiony wzorzec w konstruktorze klasy `Dict`, obejmującym w jednej definicji typu danych wszystkie techniki pozwalające tworzyć niezawodne słowniki.

```
function Dict(elements) {
    // Opcjonalnie można zastosować początkową tabelę
    this.elements = elements || {}; // Prosty obiekt
}

Dict.prototype.has = function(key) {
    // Tylko własna właściwość
    return {}.hasOwnProperty.call(this.elements, key);
};

Dict.prototype.get = function(key) {
    // Tylko własna właściwość
    return this.has(key)
        ? this.elements[key]
        : undefined;
};
```

```
Dict.prototype.set = function(key, val) {
    this.elements[key] = val;
};
```

```
Dict.prototype.remove = function(key) {
    delete this.elements[key];
};
```

**Zauważ, że implementacja metody `Dict.prototype.set` nie jest chroniona, ponieważ klucze słownika stają się właściwościami należącymi do obiektu `elements` (nawet jeśli w prototypie `Object.prototype` istnieje właściwość o tej samej nazwie).**

Przedstawiona abstrakcja jest bardziej niezawodna niż rozwiązanie oparte na domyślnej składni obiektów JavaScriptu i prawie równie wygodna w użyciu.

```
var dict = new Dict({
    alicja: 34,
    robert: 24,
    maciej: 62
});

dict.has("alicja");    // true
dict.get("robert");    // 24
dict.has("valueOf");  // false
```

Przypomnij sobie ze sposobu 44., że w niektórych środowiskach JavaScriptu nazwa specjalnej właściwości `__proto__` może prowadzić do problemów. W części środowisk właściwość `__proto__` jest dziedziczona po prototypie `Object.prototype`, dlatego puste obiekty są (na szczęście) naprawdę puste.

```
var empty = Object.create(null);
"__proto__" in empty;
// false (w niektórych środowiskach)
```

```
var hasOwn = {}.hasOwnProperty;
hasOwn.call(empty, "__proto__");
// false (w niektórych środowiskach)
```

W innych środowiskach operator `in` zwraca w takiej sytuacji wartość `true`.

```
var empty = Object.create(null);
"__proto__" in empty; // true (w niektórych środowiskach)
```

```
var hasOwn = {}.hasOwnProperty;
hasOwn.call(empty, "__proto__"); // false (w niektórych środowiskach)
```

Jednak, niestety, jeszcze inne środowiska zaśmiejają wszystkie obiekty właściwością `__proto__`.

```
var empty = Object.create(null);
"__proto__" in empty; // true (w niektórych środowiskach)
```

```
var hasOwn = {}.hasOwnProperty;
hasOwn.call(empty, "__proto__"); // true (w niektórych środowiskach)
```

To oznacza, że w zależności od środowiska poniższy kod może zwracać różne wyniki.

```
var dict = new Dict();
dict.has("__proto__"); // ?
```

Aby zapewnić maksymalną przenośność i bezpieczeństwo, trzeba do każdej metody klasy `Dict` dodać obsługę specjalnego przypadku — klucza `"__proto__"`. W efekcie powstaje bardziej skomplikowana, ale bezpieczniejsza implementacja.

```
function Dict(elements) {
    // Umożliwienie obsługi opcjonalnej początkowej tablicy
    this.elements = elements || {}; // Prosty obiekt
    this.hasSpecialProto = false;   // Czy występuje klucz "__proto__"?
    this.specialProto = undefined;  // Element "__proto__"
}

Dict.prototype.has = function(key) {
    if(key === "__proto__") {
        return this.hasSpecialProto;
    }
    // Tylko własne właściwości
    return {}.hasOwnProperty.call(this.elements, key);
};

Dict.prototype.get = function(key) {
    if (key === "__proto__") {
        return this.specialProto;
    }
    // Tylko własne właściwości
    return this.has(key)
        ? this.elements[key]
        : undefined;
};

Dict.prototype.set = function(key, val) {
    if (key === "__proto__") {
        this.hasSpecialProto = true;
        this.specialProto = val;
    } else {
        this.elements[key] = val;
    }
};

Dict.prototype.remove = function(key) {
    if (key === "__proto__") {
        this.hasSpecialProto = false;
        this.specialProto = undefined;
    } else {
        delete this.elements[key];
    }
};
```

To rozwiązanie zadziała niezależnie od tego, czy środowisko udostępnia właściwość `__proto__`, ponieważ unika używania właściwości o tej nazwie.

```
var dict = new Dict();  
dict.has("__proto__"); // false
```

### Co warto zapamiętać?

- Stosuj metodę `hasOwnProperty`, aby zabezpieczyć się przed zaśmiecaniem przez prototypy.
- Wykorzystaj zasięg leksykalny i metodę `call`, aby chronić się przed przesłonięciem metody `hasOwnProperty`.
- Pomyśl o zaimplementowaniu operacji słownika w klasie, która ukrywa szablonowe testy związane z nazwą `hasOwnProperty`.
- Wykorzystaj klasę słownika do zabezpieczenia się przed używaniem nazwy `"__proto__"` jako klucza.

### Sposób 46. Stosuj tablice zamiast słowników przy tworzeniu kolekcji uporządkowanych

Obiekt w JavaScriptcie można potraktować jak nieuporządkowaną kolekcję właściwości. Pobieranie i ustawianie różnych właściwości jest możliwe w dowolnym porządku. Niezależnie od kolejności efekt będzie taki sam, a wydajność — bardzo podobna. Standard ECMAScript nie określa kolejności zapisywania właściwości. Nie wyjaśnia też dokładnie kwestii ich enumeracji.

Warto jednak zauważyć, że pętla `for...in` musi wybrać *jakiś* porządek przy enumeracji właściwości obiektów. Ponieważ zgodnie z tym standardem silniki JavaScriptu mogą wybrać dowolną kolejność, może to wpływać na działanie programu. Często popełnianym błędem jest tworzenie interfejsu API, który wymaga obiektu reprezentującego uporządkowane odwzorowanie łańcuchów znaków na wartości (co może być potrzebne na przykład przy tworzeniu uporządkowanego raportu).

```
function report(highScores) {  
    var result = "";  
    var i = 1;  
    for (var name in highScores) { // Kolejność jest nieprzewidywalna  
        result += i + ". " + name + ": " +  
            highScores[name] + "\n";  
        i++;  
    }  
    return result;  
}
```

```
report([ { name: "Henryk", points: 1110100 },
         { name: "Stefan", points: 1064500 },
         { name: "Bartek", points: 1050200 } ]);
// ?
```

Ponieważ w poszczególnych środowiskach właściwości obiektu są zapisywane i wyświetlane w odmiennej kolejności, przedstawiona funkcja może zwracać różne łańcuchy znaków. Może to zakłócić porządek w raporcie z najlepszymi wynikami.

Pamiętaj, że nie zawsze jest oczywiste, czy program zależy od kolejności, w jakiej właściwości obiektu są wyświetlane. Jeśli nie sprawdzisz programu w kilku środowiskach JavaScriptu, możesz nawet nie zauważyć, że jego działanie zmienia się w zależności od porządku elementów w pętli `for...in`.

Jeśli program wymaga określonej kolejności elementów w strukturze danych, zastosuj tablicę zamiast słownika. Pokazana wcześniej funkcja `report` będzie działać przewidywalnie w dowolnym środowisku JavaScriptu, jeśli w jej interfejsie API użyjesz tablicy obiektów zamiast jednego obiektu.

```
function report(highScores) {
    var result = "";
    for (var i = 0, n = highScores.length; i < n; i++) {
        var score = highScores[i];
        result += (i + 1) + ". " +
            score.name + ": " + score.points + "\n";
    }
    return result;
}

report([ { name: "Henryk", points: 1110100 },
         { name: "Stefan", points: 1064500 },
         { name: "Bartek", points: 1050200 } ]);
// "1. Henryk: 1110100\n2. Stefan: 1064500\n3. Bartek: 1050200\n"
```

Ta wersja przyjmuje tablicę obiektów (każdy z nich ma właściwości `name` i `points`), dzięki czemu w przewidywalny sposób przechodzi po elementach w ściśle określonej kolejności — od elementu zerowego do `highScores.length - 1`.

Zależność od kolejności często jest odczuwalna w arytmetyce zmiennoprzecinkowej. Przyjrzyj się słownikowi z tytułami i ocenami filmów.

```
var ratings = {
    "Good Will Hunting": 0.8,
    "Mystic River": 0.7,
    "21": 0.6,
    "Doubt": 0.9
};
```

W sposobie 2. wyjaśniono, że arytmetyka zmiennoprzecinkowa daje czasem wyniki zależne od kolejności wykonywania operacji. W połączeniu z niezdefiniowaną kolejnością enumeracji może to skutkować nieprzewidywalnym działaniem pętli.

```
var total = 0, count = 0;
for (var key in ratings) { // Nieprzewidywalna kolejność
    total += ratings[key];
    count++;
}
total /= count;
total; // ?
```

Okazuje się, że popularne środowiska JavaScriptu wykonują tę pętlę w różnej kolejności. Niektóre środowiska pobierają klucze z obiektu w kolejności ich występowania w kodzie. Obliczenia wyglądają wtedy tak:

```
(0.8 + 0.7 + 0.6 + 0.9) / 4 // 0.75
```

Inne środowiska zawsze pobierają najpierw potencjalne indeksy tablicy, a dopiero potem pozostałe klucze. Ponieważ film *21* ma tytuł, który może być indeksem tablicy, jest pobierany jako pierwszy. Wykonywana jest wtedy następująca operacja:

```
(0.6 + 0.8 + 0.7 + 0.9) / 4 // 0.7499999999999999
```

W takiej sytuacji lepiej jest zastosować w słowniku wartości całkowitoliczbowe, ponieważ liczby całkowite można dodawać w dowolnej kolejności. W tym podejściu narażone na błędy operacje dzielenia są wykonywane na samym końcu. Co ważne, ma to miejsce po zakończeniu pracy pętli.

```
(8 + 7 + 6 + 9) / 4 / 10 // 0.75
(6 + 8 + 7 + 9) / 4 / 10 // 0.75
```

Przy wykonywaniu operacji w pętli `for...in` zawsze należy zwracać uwagę na to, aby niezależnie od kolejności działały one tak samo.

### Co warto zapamiętać?

- Unikaj polegania na kolejności, w jakiej pętla `for...in` pobiera właściwości obiektu.
- Jeśli agregujesz dane ze słownika, upewnij się, że wykonywana operacja jest niezależna od kolejności.
- Dla kolekcji uporządkowanych stosuj tablice zamiast słowników.

### Sposób 47. Nigdy nie dodawaj enumerowanych właściwości do prototypu `Object.prototype`

Pętla `for...in` jest niezwykle wygodna w użyciu, jednak (co opisano w sposobie 43.) jest narażona na zaśmiecanie przez prototypy. Tę pętlę zdecydowanie najczęściej stosuje się do enumerowania elementów słownika. Wniosek z tego

jest oczywisty — jeśli chcesz umożliwić stosowanie pętli `for...in` dla obiektów reprezentujących słowniki, nigdy nie dodawaj enumerowanych właściwości do współużytkowanego prototypu `Object.prototype`.

Ta reguła może rozczarowywać. Cóż może być wygodniejszego od dodawania do prototypu `Object.prototype` metod pomocniczych dostępnych później we wszystkich obiektach? Może to być na przykład metoda `allKeys` zwracająca tablicę nazw właściwości obiektu.

```
Object.prototype.allKeys = function() {  
  var result = [];  
  for (var key in this) {  
    result.push(key);  
  }  
  return result;  
};
```

Niestety, ta metoda zaśmieca nawet zwracane przez siebie wyniki.

```
({ a: 1, b: 2, c: 3 }).allKeys(); // ["allKeys", "a", "b", "c"]
```

Oczywiście zawsze można poprawić metodę `allKeys`, aby ignorowała właściwości prototypu `Object.prototype`. Jednak możliwości związane są z odpowiedzialnością — manipulacje współużytkowanym prototypem mają wpływ na wszystkich użytkowników obiektu. Dodanie jednej właściwości do prototypu `Object.prototype` zmusza *wszystkich, aby w każdym miejscu zabezpieczali pętle `for...in`*.

Mniej wygodne, ale ułatwiające pisanie innego kodu rozwiązanie polega na zdefiniowaniu `allKeys` jako funkcji, a nie jako metody.

```
function allKeys(obj) {  
  var result = [];  
  for (var key in obj) {  
    result.push(key);  
  }  
  return result;  
}
```

Jeśli chcesz dodać właściwości do prototypu `Object.prototype` w sposób mniej szkodliwy dla innego kodu, standard ES5 zapewnia odpowiedni mechanizm. Metoda `Object.defineProperty` pozwala zdefiniować właściwość obiektu razem z metadanymi — **atrybutami** właściwości. Można na przykład zdefiniować właściwość w identyczny sposób jak wcześniej, ale sprawić, że będzie ona niewidoczna dla pętli `for...in`. Wymaga to ustawienia atrybutu `enumerable` na wartość `false`.

```
Object.defineProperty(Object.prototype, "allKeys", {  
  value: function() {  
    var result = [];  
    for (var key in this) {  
      result.push(key);  
    }  
  }  
});
```

```
        return result;
    },
    writable: true,
    enumerable: false,
    configurable: true
  });
```

To prawda, ten kod jest rozwlekły. Jednak zaletą tej wersji jest to, że nie zaśmieca wszystkich pętli `for...in` w każdym egzemplarzu typu `Object`.

Tę technikę warto stosować także w innych obiektach. Gdy potrzebujesz właściwości, która ma być niewidoczna w pętlach `for...in`, metoda `Object.defineProperty` Ci pomoże.

### Co warto zapamiętać?

- Unikaj dodawania właściwości do prototypu `Object.prototype`.
- Pomyśl o pisaniu funkcji zamiast metod prototypu `Object.prototype`.
- Jeśli dodajesz właściwości do prototypu `Object.prototype`, użyj metody `Object.defineProperty` ze standardu ES5, aby zdefiniować je jako właściwości nieenumerowane.

## Sposób 48. Unikaj modyfikowania obiektu w trakcie enumeracji

W sieci społecznościowej przechowywany jest zbiór użytkowników. Dla każdego z nich zarejestrowana jest lista znajomych.

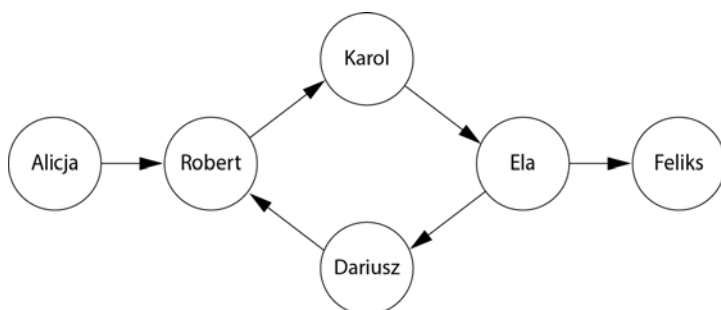
```
function Member(name) {
    this.name = name;
    this.friends = [];
}

var a = new Member("Alicja"),
    b = new Member("Robert"),
    c = new Member("Karol"),
    d = new Member("Dariusz"),
    e = new Member("Ela"),
    f = new Member("Feliks");
```

```
a.friends.push(b);
b.friends.push(c);
c.friends.push(e);
d.friends.push(b);
e.friends.push(d, f);
```

Przeszukiwanie tej sieci wymaga przejścia po grafie sieci społecznościowej (rysunek 5.1). Tę operację często implementuje się za pomocą zbiorów roboczych.





**Rysunek 5.1.** Graf sieci społecznościowej

Należy rozpocząć od korzenia, dodawać wykryte węzły i usuwać odwiedzone. Dobrym pomysłem może wydawać się próba zaimplementowania tej operacji za pomocą jednej pętli `for...in`.

```
Member.prototype.inNetwork = function(other) {  
    var visited = {};  
    var workset = {};  
  
    workset[this.name] = this;  
  
    for (var name in workset) {  
        var member = workset[name];  
        delete workset[name];    // Modyfikacja w trakcie enumeracji  
        if (name in visited) {    // Nie należy ponownie odwiedzać węzłów  
            continue;  
        }  
        visited[name] = member;  
        if (member === other) {    // Znalezione?  
            return true;  
        }  
        member.friends.forEach(function(friend) {  
            workset[friend.name] = friend;  
        });  
    }  
  
    return false;  
};
```

Niestety, w wielu środowiskach JavaScriptu ten kod w ogóle nie zadziała.

```
a.inNetwork(f);    // false
```

Dlaczego tak się dzieje? Okazuje się, że pętla `for...in` nie musi na bieżąco sprawdzać modyfikacji w enumerowanych obiektach. Zgodnie ze standardem ECMA-Script środowiska JavaScriptu mogą w różny sposób uwzględniać wprowadzane modyfikacje. Oto fragment standardu:

Jeśli w trakcie enumeracji do obiektu dodawane są nowe właściwości, nie ma gwarancji, że zostaną one uwzględnione w czasie bieżącej enumeracji.

Ta nieprecyzyjna specyfikacja powoduje, że nie można oczekiwać, iż pętla `for...in` będzie działać w przewidywalny sposób, gdy enumerowany obiekt zostanie zmodyfikowany.

Oto inna próba przejścia przez graf — tym razem z samodzielnym zarządzaniem pętlą. Wykorzystano tu klasę słownika, aby uniknąć zjawiska zaśmiecania przez prototyp. Słownik można zapisać w klasie `WorkSet`, rejestrującej liczbę elementów przechowywanych obecnie w zbiorze.

```
function WorkSet() {
    this.entries = new Dict();
    this.count = 0;
}

WorkSet.prototype.isEmpty = function() {
    return this.count === 0;
};

WorkSet.prototype.add = function(key, val) {
    if (this.entries.has(key)) {
        return;
    }
    this.entries.set(key, val);
    this.count++;
};

WorkSet.prototype.get = function(key) {
    return this.entries.get(key);
};

WorkSet.prototype.remove = function(key) {
    if (!this.entries.has(key)) {
        return;
    }
    this.entries.remove(key);
    this.count--;
};
```

Aby można było wybrać dowolny element zbioru, potrzebna jest nowa metoda w klasie `Dict`.

```
Dict.prototype.pick = function() {
    for (var key in this.elements) {
        if (this.has(key)) {
            return key;
        }
    }
    throw new Error("empty dictionary");
};
```

```
WorkSet.prototype.pick = function() {  
    return this.entries.pick();  
};
```

Teraz można zaimplementować metodę `inNetwork` za pomocą prostej pętli `while`, wybierając pojedynczo dowolne elementy i usuwać je ze zbioru roboczego.

```
Member.prototype.inNetwork = function(other) {  
    var visited = {};  
    var workset = new WorkSet();  
    workset.add(this.name, this);  
    while (!workset.isEmpty()) {  
        var name = workset.pick();  
        var member = workset.get(name);  
        workset.remove(name);  
        if (name in visited) { // Nie należy ponownie odwiedzać węzłów  
            continue;  
        }  
        visited[name] = member;  
        if (member === other) { // Znaleziono?  
            return true;  
        }  
        member.friends.forEach(function(friend) {  
            workset.add(friend.name, friend);  
        });  
    }  
    return false;  
};
```

Metoda `pick` jest **niedeterministyczna**. Oznacza to, że według semantyki języka nie gwarantuje jednego, przewidywalnego wyniku. Wynika to z tego, że w różnych środowiskach JavaScriptu (a nawet — przynajmniej teoretycznie — w różnych przebiegach w tym samym środowisku) pętla `for...in` może wybrać inną kolejność enumeracji. Używanie niedeterministycznych operacji bywa skomplikowane, ponieważ wprowadzają one aspekt nieprzewidywalności do programu. Testy, które na jednej platformie kończą się sukcesem, na innej mogą kończyć się niepowodzeniem. Możliwe nawet, że na tej samej platformie test raz zakończy się powodzeniem, a raz porażką.

Niektórych przyczyn niedeterminizmu nie da się zlikwidować. Losowy generator liczb *powinien* generować nieprzewidywalne wyniki. Bieżąca data i godzina za każdym razem będą inne. Reakcje na działania użytkowników, na przykład kliknięcia myszą i wciśnięcia klawiszy, będą inne w zależności od użytkownika. Warto jednak jednoznacznie określić, w których fragmentach programu oczekiwany jest konkretny wynik, a w których mogą występować zmiany.

Dlatego warto rozważyć deterministyczną alternatywę dla algorytmu oparte-go na zbiorze roboczym. Jest to algorytm oparty na liście roboczej. Jeśli zapiszesz elementy w tablicy zamiast w zbiorze, metoda `inNetwork` zawsze będzie przechodzić przez graf w dokładnie tej samej kolejności.

```
Member.prototype.inNetwork = function(other) {
  var visited = {};
  var worklist = [this];
  while (worklist.length > 0) {
    var member = worklist.pop();
    if (member.name in visited) { // Nie należy ponownie odwiedzać elementów
      continue;
    }
    visited[member.name] = member;
    if (member === other) {      // Znaleziono?
      return true;
    }
    member.friends.forEach(function(friend) {
      worklist.push(friend);    // Dodawanie do listy roboczej
    });
  }
  return false;
};
```

Ta wersja metody `inNetwork` dodaje i usuwa elementy w deterministyczny sposób. Ponieważ ta metoda zawsze zwraca wartość `true` dla powiązanych użytkowników niezależnie od ścieżki, jaką ich połączy, wynik końcowy jest zawsze taki sam. Jednak nie musi to być prawdą dla innych metod, na przykład dla wersji metody `inNetwork` zwracającej ścieżkę znaną w trakcie poruszania się między elementami grafu.

### Co warto zapamiętać?

- Pamiętaj, aby nie modyfikować obiektu w trakcie enumeracji jego właściwości w pętli `for...in`.
- Zastosuj pętlę `while` lub klasyczną pętlę `for` zamiast pętli `for...in` na potrzeby przechodzenia po zawartości obiektu, która może się zmieniać w pętli.
- Aby zapewnić przewidywalną enumerację dla zmieniającej się struktury danych, pomyśl o wykorzystaniu sekwencyjnej struktury danych (na przykład tablicy zamiast słownika).

### Sposób 49. Stosuj pętlę `for` zamiast pętli `for...in` przy przechodzeniu po tablicy

Jaka będzie wartość zmiennej `mean` w poniższym kodzie?

```
var scores = [98, 74, 85, 77, 93, 100, 89];
var total = 0;
for (var score in scores) {
  total += score;
}
var mean = total / scores.length;
mean; // ?
```

Czy zauważyłeś błąd? Jeśli stwierdziłeś, że wartość zmiennej to „88”, zrozumiałeś przeznaczenie programu, jednak w rzeczywistości wynik będzie inny. W tym programie znalazł się bardzo łatwy do popełnienia błąd — pomyłono *klucze* z *wartościami* w tablicy liczb. Pętla `for...in` w enumeracji zawsze używa kluczy. Interesująca następna próba to  $(0+1+\dots+6)/7 = 21/7 = 3$ , jednak także ta odpowiedź jest nieprawidłowa. Pamiętaj, że klucze właściwości w obiektach to łańcuchy znaków. Dotyczy to nawet indeksowanych właściwości tablicy. Dlatego operacja `+=` łączy łańcuchy znaków i zwraca niezgodną z oczekiwaniami wartość „00123456”. Efekt końcowy to zaskakująca wartość zmiennej `mean` równa 17636.571428571428.

Właściwy sposób przechodzenia po zawartości tablicy to zastosowanie klasycznej pętli `for`.

```
var scores = [98, 74, 85, 77, 93, 100, 89];
var total = 0;
for (var i = 0, n = scores.length; i < n; i++) {
    total += scores[i];
}
var mean = total / scores.length;
mean; // 88
```

To podejście gwarantuje, że w razie potrzeby dostępne są indeksy całkowitoliczbowe i wartości elementów tablicy. Te wartości nie mylą się ze sobą, a kod nie powoduje nieoczekiwanego przekształcania liczb na łańcuchy znaków. Ponadto zastosowane rozwiązanie sprawia, że iteracja przebiega we właściwej kolejności, a także chroni przed dodaniem niecałkowitoliczbowych właściwości przechowywanych w obiekcie reprezentującym tablicę lub w łańcuchu jego prototypów.

Zwróć uwagę na zastosowanie w pętli `for` zmiennej `n` reprezentującej długość tablicy. Jeśli w ciele pętli `tablica` nie jest modyfikowana, działanie przedstawionego wcześniej kodu jest identyczne jak przy ponownym obliczaniu długości tablicy w każdej iteracji.

```
for (var i = 0; i < scores.length; i++) { ... }
```

Jednak jednokrotne obliczenie długości tablicy na początku pętli zapewnia kilka niewielkich korzyści. Nawet kompilatory optymalizujące kod w JavaScriptcie często mają trudności z udowodnieniem, że można bezpiecznie zrezygnować z ponownego obliczania wartości `scores.length`. Jeszcze ważniejszą zaletą pierwotnej wersji jest to, że informuje ona czytelnika kodu o tym, że warunek zakończenia pracy pętli jest prosty i niezmienny.

### Co warto zapamiętać?

- Zawsze stosuj pętlę `for` zamiast pętli `for...in` do przechodzenia po indeksowanych właściwościach tablicy.

- Pomyśl o zapisaniu wartości właściwości `length` tablicy w zmiennej lokalnej przed pętlą, aby uniknąć ponownych obliczeń w samej pętli.

## Sposób 50. Zamiast pętli stosuj metody do obsługi iteracji

Dobrzy programiści nie lubią pisać dwa razy tego samego kodu. Kopiowanie i wklejanie szablonowego kodu powoduje duplikację błędów, utrudnia modyfikowanie programów, zaśmieca aplikacje powtarzającymi się wzorcami i wymaga od programistów nieustannego wymyślania koła od nowa. Prawdopodobnie najgorsze jest jednak to, że powtórzenia utrudniają czytelnikom kodu dostrzeżenie drobnych różnic między poszczególnymi wystąpieniami danego wzorca.

Pętle `for` w JavaScriptcie są zwięzłe i znane użytkownikom wielu innych języków (takich jak C, Java i C#), jednak działają nieco inaczej w zależności od składni. Niektóre z najczęściej występujących błędów programistycznych to efekt prostych pomyłek przy określaniu warunku zakończenia pracy pętli.

```
for (var i = 0; i <= n; i++) { ... }  
// Dodatkowa końcowa iteracja  
for (var i = 1; i < n; i++) { ... }  
// Pominięcie pierwszej iteracji  
for (var i = n; i >= 0; i--) { ... }  
// Dodatkowa początkowa iteracja  
for (var i = n - 1; i > 0; i--) { ... }  
// Pominięcie ostatniej iteracji
```

Prawda jest taka, że ustalanie warunków wyjścia z pętli to prawdziwa męczarnia. Jest to nudne i można popełnić przy tym wiele drobnych błędów.

Na szczęście JavaScript udostępnia domknięcia (zobacz sposób 11.), które są wygodnym i zwięzłym narzędziem do budowania abstrakcji do iterowania odpowiadających pokazanym wzorcom. To rozwiązanie pozwala uniknąć kopiowania i wklejania nagłówków pętli.

Standard ES5 udostępnia metody pomocnicze związane z najczęściej używanymi wzorcami. Najprostsza z tych metod to `Array.prototype.forEach`. Dzięki niej zamiast pisać:

```
for (var i = 0, n = players.length; i < n; i++) {  
    players[i].score++;  
}
```

można napisać:

```
players.forEach(function(p) {  
    p.score++;  
});
```

Ten kod nie tylko jest bardziej zwięzły i czytelny, ale też nie wymaga podawania warunków wyjścia z pętli *ani* indeksów tablicy.

Inny często występujący wzorzec związany jest z budowaniem nowej tablicy w wyniku wykonania określonych operacji na każdym elemencie innej tablicy. To zadanie można wykonać w pętli.

```
var trimmed = [];
for (var i = 0, n = input.length; i < n; i++) {
    trimmed.push(input[i].trim());
}
```

Inna możliwość to zastosowanie instrukcji `forEach`.

```
var trimmed = [];
input.forEach(function(s) {
    trimmed.push(s.trim());
});
```

Jednak wzorzec budowania nowej tablicy na podstawie istniejącej jest wykorzystywany tak często, że w standardzie ES5 dodano metodę `Array.prototype.map`, aby uprościć omawiane zadanie i umożliwić jego wykonywanie w bardziej elegancki sposób.

```
var trimmed = input.map(function(s) {
    return s.trim();
});
```

Inny często stosowany wzorzec to tworzenie nowej tablicy zawierającej tylko wybrane elementy istniejącej. Dzięki metodzie `Array.prototype.filter` jest to proste. Przyjmuje ona **predykat**, czyli funkcję zwracającą wartość `true` dla elementów, które należy zachować w nowej tablicy, i `false` dla pomijanych elementów. Można na przykład pobrać z listy tylko elementy o cenie z określonego przedziału.

```
listings.filter(function(listing) {
    return listing.price >= min && listing.price <= max;
});
```

Oczywiście to tylko metody dostępne domyślnie w standardzie ES5. Nic nie stoi na przeszkodzie, aby samodzielnie zdefiniować własne abstrakcje do obsługi iteracji. Czasem trzeba na przykład pobrać z tablicy najdłuższy przedrostek spełniający określony predykat.

```
function takeWhile(a, pred) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
        if (!pred(a[i], i)) {
            break;
        }
        result[i] = a[i];
    }
    return result;
}

var prefix = takeWhile([1, 2, 4, 8, 16, 32], function(n) {
    return n < 10;
}); // [1, 2, 4, 8]
```

Zauważ, że kod przekazuje indeks `i` tablicy do funkcji `pred`, która określa, czy dany element należy wykorzystać, czy zignorować. Wszystkie funkcje do obsługi iteracji występujące w bibliotece standardowej (w tym `forEach`, `map` i `filter`) przekazują indeks tablicy do funkcji podanej przez użytkownika.

Można też zdefiniować `takeWhile` jako metodę prototypu `Array.prototype`. W sposobie 42. znajdziesz omówienie skutków stosowania techniki monkey patching dla standardowych prototypów takich jak `Array.prototype`.

```
Array.prototype.takeWhile = function(pred) {
  var result = [];
  for (var i = 0, n = this.length; i < n; i++) {
    if (!pred(this[i], i)) {
      break;
    }
    result[i] = this[i];
  }
  return result;
};

var prefix = [1, 2, 4, 8, 16, 32].takeWhile(function(n) {
  return n < 10;
}); // [1, 2, 4, 8]
```

Pętle mają jednak pewną zaletę w porównaniu z funkcjami do obsługi iteracji. Chodzi o instrukcje sterowania przepływem, takie jak `break` i `continue`. Próba zaimplementowania funkcji `takeWhile` za pomocą metody `forEach` prowadzi do dziwnego kodu.

```
function takeWhile(a, pred) {
  var result = [];
  a.forEach(function(x, i) {
    if (!pred(x)) {
      // ?
    }
    result[i] = x;
  });
  return result;
}
```

Można zastosować wewnętrzny wyjątek, aby umożliwić wczesne wychodzenie z pętli, jest to jednak dziwaczne i (prawdopodobnie) niewydajne rozwiązanie.

```
function takeWhile(a, pred) {
  var result = [];
  var earlyExit = {}; // Unikatowa wartość sygnalizująca wyjście z pętli
  try {
    a.forEach(function(x, i) {
      if (!pred(x)) {
        throw earlyExit;
      }
      result[i] = x;
    });
  } catch(e) {
```



```

    if (e !== earlyExit) { // Przechwytywanie tylko wyjątku earlyExit
      throw e;
    }
  }
  return result;
}

```

Gdy abstrakcja okazuje się bardziej rozwlekła niż zastępowany przez nią kod, oznacza to, że lekarstwo stało się gorsze od choroby.

Do tworzenia pętli, które mogą wcześniej kończyć pracę, można też wykorzystać metody tablic ze standardu ES5 — `some` i `every`. Warto przy tym zauważyć, że nie po to je utworzono. Służą one do stosowania wywoływanego zwrotnie predykatu do każdego elementu tablicy. Metoda `some` zwraca wartość logiczną informującą, czy wywołanie zwrotne zwróciło wartość `true` dla któregośkolwiek elementu tablicy.

```

[1, 10, 100].some(function(x) { return x > 5; }); // true
[1, 10, 100].some(function(x) { return x < 0; }); // false

```

Metoda `every` zwraca wartość logiczną określającą, czy wywołanie zwrotne zwróciło wartość `true` dla wszystkich elementów.

```

[1, 2, 3, 4, 5].every(function(x) { return x > 0; }); // true
[1, 2, 3, 4, 5].every(function(x) { return x < 3; }); // false

```

Obie te metody stosują **przetwarzanie skrócone**. Jeśli wywołanie zwrotne w metodzie `some` zwróci choć jedną wartość `true`, metoda kończy pracę bez przetwarzania pozostałych elementów. Podobnie metoda `every` natychmiast kończy działanie, gdy wywołanie zwrotne zwróci pierwszą wartość `false`.

Ta cecha sprawia, że omawiane metody są przydatne jako wersja metody `forEach` z możliwością wczesnego kończenia pracy. Poniżej pokazano, jak zaimplementować funkcję `takeWhile` za pomocą metody `every`.

```

function takeWhile(a, pred) {
  var result = [];
  a.every(function(x, i) {
    if (!pred(x)) {
      return false; // Odpowiednik instrukcji break
    }
    result[i] = x;
    return true; // Odpowiednik instrukcji continue
  });
  return result;
}

```

## Co warto zapamiętać?

- Stosuj metody do obsługi iteracji, na przykład `Array.prototype.forEach` i `Array.prototype.map`, zamiast pętli `for`, aby zwiększyć czytelność kodu i uniknąć powtarzania kodu sterującego pętlą.

- Wykorzystaj niestandardowe funkcje do obsługi iteracji, aby implementować często stosowane wzorce związane z pętlami, niedostępne w bibliotece standardowej.
- Standardowe pętle są przydatne w sytuacjach, gdy czasem konieczne jest przedwczesne wyjście z pętli. Podobny efekt można uzyskać dzięki metodom `some` i `every`.

## Sposób 51. Wykorzystaj uniwersalne metody klasy `Array` w obiektach podobnych do tablic

Standardowe metody prototypu `Array.prototype` zaprojektowano po to, aby można je było wykorzystać jako metody innych obiektów (także tych, które nie dziedziczą po klasie `Array`). Okazuje się, że tego rodzaju obiekty pojawiają się w JavaScriptcie w rozmaitych miejscach.

Dobrym przykładem jest obiekt `arguments` funkcji, opisany w sposobie 22. Obiekt ten nie dziedziczy po prototypie `Array.prototype`, dlatego nie wystarczy wywołanie metody `arguments.forEach`, aby przejść po wszystkich argumentach funkcji. Zamiast tego trzeba pobrać referencję do obiektu z metodą `forEach` i wykorzystać metodę `call` (zobacz sposób 20.).

```
function highlight() {  
    [].forEach.call(arguments, function(widget) {  
        widget.setBackground("yellow");  
    });  
}
```

Metoda `forEach` to obiekt typu `Function`, co oznacza, że dziedziczy metodę `call` po prototypie `Function.prototype`. Dlatego można wywołać metodę `forEach` z niestandardową wartością dla określonego obiektu związanego z `this` (tu używany jest obiekt `arguments`), po którym podawana jest dowolna liczba argumentów (tu argumentem jest jedna wywoływana zwrótnie funkcja). Ten kod działa zgodnie z oczekiwaniami.

W przeglądarkach internetowych obiektem podobnym do tablic jest klasa `NodeList` z modelu DOM. Operacje takie jak `document.getElementsByTagName`, które pobierają węzły strony internetowej, zwracają wyniki wyszukiwania w postaci obiektów typu `NodeList`. Te obiekty (podobnie jak obiekty typu `arguments`) działają podobnie jak tablice, ale nie dziedziczą po prototypie `Array.prototype`.

Co oznacza stwierdzenie, że obiekt jest „podobny do tablicy”? Podstawowy kontrakt określa, że tablice muszą być zgodne z dwoma prostymi regułami.

- Muszą mieć całkowitoliczbową właściwość `length` o wartości z przedziału od 0 do  $2^{32}-1$ .

- Właściwość `length` musi mieć wartość większą niż największy *indeks* obiektu. Indeks to liczba całkowita z przedziału od 0 do  $2^{32}-2$ , której postać tekstowa jest kluczem właściwości obiektu.

Te cechy wystarczą, aby obiekt był zgodny z metodami prototypu `Array.prototype`. Nawet prosty literal obiektowy można wykorzystać do utworzenia obiektu podobnego do tablicy.

```
var arrayLike = { 0: "a", 1: "b", 2: "c", length: 3};
var result = Array.prototype.map.call(arrayLike, function(s) {
    return s.toUpperCase();
}); // ["A", "B", "C"]
```

Także łańcuchy znaków działają jak niezmiennicze tablice, ponieważ można stosować w nich indeksy i sprawdzać długość za pomocą właściwości `length`. Dlatego metody prototypu `Array.prototype`, które nie modyfikują tablicy, działają także dla łańcuchów znaków.

```
var result = Array.prototype.map.call("abc", function(s) {
    return s.toUpperCase();
}); // ["A", "B", "C"]
```

Jednak zasymulowanie *wszystkich* operacji tablic z JavaScriptu jest bardziej skomplikowane. Wynika to z dwóch innych aspektów działania tablic.

- Ustawienie właściwości `length` na wartość  $n$  mniejszą od obecnej automatycznie powoduje usunięcie właściwości o indeksach równych  $n$  lub większych od tej wartości.
- Dodanie właściwości o indeksie  $n$  większym lub równym wartości właściwości `length` prowadzi do automatycznego ustawienia właściwości `length` na wartość  $n+1$ .

Trudności sprawia zwłaszcza druga z tych reguł, ponieważ wymaga śledzenia dodawania indeksowanych właściwości w celu automatycznego aktualizowania wartości właściwości `length`. Na szczęście obsługa żadnej z tych reguł nie jest konieczna do używania metod prototypu `Array.prototype`, ponieważ wszystkie te metody aktualizują właściwość `length` w momencie dodawania lub usuwania indeksowanych właściwości.

Istnieje tylko jedna metoda klasy `Array`, która nie jest w pełni uniwersalna. Jest to metoda `concat` służąca do złączania tablic. Tę metodę można wywołać dla dowolnego odbiorcy podobnego do tablic, jednak sprawdza ona wartość właściwości `[[Class]]` argumentów. Jeśli argument rzeczywiście jest tablicą, jego zawartość jest dołączana do wyniku. W przeciwnym razie argument jest dodawany jako pojedynczy element. To oznacza, że nie można na przykład złączyć tablicy z zawartością obiektu `arguments`.

```
function namesColumn() {
    return ["Names"].concat(arguments);
}
```

```

}
namesColumn("Alicja", "Robert", "Maciej");
// ["Names", {0: "Alicja", 1: "Robert", 2: "Maciej"}]

```

Aby metoda `concat` traktowała obiekty podobne do tablic jak prawdziwe tablice, trzeba je samodzielnie przekształcić. Popularny i zwięzły idiom, który to umożliwia, to użycie metody `slice` dla obiektu podobnego do tablicy.

```

function namesColumn() {
    return ["Names"].concat([].slice.call(arguments));
}
namesColumn("Alicja", "Robert", "Maciej");
// ["Names", "Alicja", "Robert", "Maciej"]

```

### Co warto zapamiętać?

- Używaj uniwersalnych metod klasy `Array` dla obiektów podobnych do tablic. W tym celu pobierz obiekty reprezentujące odpowiednie metody i użyj ich metody `call`.
- Uniwersalne metody klasy `Array` można stosować do obiektów, które mają indeksowane właściwości i odpowiednią właściwość `length`.

## Sposób 52. Przedkładaj literały tablicowe nad konstruktor klasy `Array`

Elegancja JavaScriptu wynika w dużej części z opartej na literałach zwięzłej składni tworzenia najczęściej używanych cegiełek programów w tym języku: obiektów, funkcji i tablic. Literał to świetny sposób na przedstawienie tablicy.

```
var a = [1, 2, 3, 4, 5];
```

Zamiast literału można też wykorzystać konstruktor klasy `Array`.

```
var a = new Array(1, 2, 3, 4, 5);
```

Nawet jeśli pominąć kwestie estetyczne, okazuje się, że z konstruktorem klasy `Array` związane są pewne problemy. Jednym z nich jest konieczność upewnienia się, że nikt nie związał zmiennej `Array` w nietypowy sposób.

```

function f(Array) {
    return new Array(1, 2, 3, 4, 5);
}
f(String); // new String(1)

```

Ponadto trzeba zadbać o to, aby nikt nie zmodyfikował globalnej zmiennej `Array`.

```

Array = String;
new Array(1, 2, 3, 4, 5); // new String(1)

```

Jest jeszcze jeden kłopotliwy specjalny przypadek. Jeśli wywołasz konstruktor klasy `Array` z jednym liczbowym argumentem, konstruktor zadziała w nietypowy sposób — spróbuje utworzyć tablicę bez elementów z właściwością `length` o wartości równej podanemu argumentowi. To oznacza, że instrukcje `["witaj"]` i `new Array("witaj")` działają tak samo, natomiast polecenia `[17]` i `new Array(17)` wykonują zupełnie odmienne operacje.

Opanowanie tych reguł nie jest trudne, jednak literały tablicowe dzięki spójniejszej semantyce są bardziej przejrzyste i mniej narażone na błędy.

### Co warto zapamiętać?

- Konstruktor klasy `Array` działa inaczej, gdy jego pierwszy argument to liczba.
- Stosuj literały tablicowe zamiast konstruktora klasy `Array`.



# 6

## Projekty bibliotek i interfejsów API

Każdy programista bywa czasem projektantem interfejsu API. Możliwe, że nie planuje napisać od razu następnej popularnej biblioteki JavaScriptu. Jednak gdy programista przez długi czas korzysta z danej platformy, rozwija zestaw rozwiązań często spotykanych problemów i wcześniej lub później zaczyna tworzyć narzędzia i komponenty wielokrotnego użytku. Nawet jeśli programista nie udostępnia ich jako niezależnych bibliotek, rozwinięcie umiejętności przydatnych autorowi bibliotek pomoże mu w pisaniu lepszych komponentów.

Projektowanie bibliotek to skomplikowane zadanie. Jest to w równej mierze sztuka, jak i nauka. To niezwykle ważne zadanie. Interfejsy API tworzą podstawowy słownik programisty. Dobrze zaprojektowany interfejs API pozwala użytkownikom (w tym zwykle samemu programiście) pisać przejrzyste, zwarte i jednoznaczne programy.

### Sposób 53. Przestrzegaj spójnych konwencji

Niewiele decyzji w większym stopniu wpływa na użytkowników interfejsu API niż konwencje tworzenia nazw i sygnatur funkcji. Konwencje są niezwykle ważne. Wyznaczają podstawowy słownik i idiomy stosowane w docelowych aplikacjach. Użytkownicy biblioteki muszą nauczyć się czytać i pisać kod za pomocą tych idiomów. Zadanie autora interfejsów polega na tym, aby ułatwić innym proces nauki. Niespójności powodują, że trudniej jest zapamiętać, które konwencje należy stosować w poszczególnych sytuacjach. Wtedy więcej czasu zajmuje sprawdzanie dokumentacji biblioteki niż wykonywanie zadań.

Jedna z najważniejszych konwencji dotyczy kolejności argumentów. Na przykład w bibliotekach do tworzenia interfejsu użytkownika zwykle występują funkcje, które przyjmują kilka wymiarów (takich jak szerokość i wysokość).

Wyświadczyć użytkownikom biblioteki przysługę i zadbać o to, aby wymiary zawsze występowały w tej samej kolejności. Warto wybrać porządek zgodny z innymi bibliotekami. Prawie wszędzie najpierw podawana jest szerokość, a następnie wysokość.

```
var widget = new Widget(320, 240); // Szerokość: 320, wysokość: 240
```

O ile nie masz naprawdę dobrego powodu do odejścia od uniwersalnie stosowanego podejścia, trzymaj się znanych rozwiązań. Jeśli biblioteka służy do tworzenia witryn internetowych, pamiętaj, że programiści zwykle posługują się kilkoma językami (takimi jak HTML, CSS i JavaScript — *to minimum*). Nie utrudniaj tym osobom życia przez niepotrzebne odchodzenie od konwencji stosowanych przez programistów w codziennej pracy. Na przykład gdy w języku CSS przyjmowane są parametry opisujące cztery krawędzie prostokąta, należy podawać je zgodnie z ruchem wskazówek zegara, rozpoczynając od góry (góra, prawa, dół, lewa). Dlatego gdy piszesz bibliotekę z podobnym interfejsem API, wykorzystaj tę kolejność. Użytkownicy będą Ci za to wdzięczni. Możliwe, że nawet nie zwrócą uwagi na zastosowaną kolejność — tym lepiej. Możesz mieć jednak pewność, że *na pewno* zauważą odejście od standardowych konwencji.

Jeśli w interfejsie API wykorzystujesz obiekty z opcjami (zobacz sposób 55.), możesz uniknąć zależności od kolejności argumentów. W przypadku standardowych opcji (takich jak szerokość i wysokość) należy zdecydować się na określoną konwencję tworzenia nazw i konsekwentnie ją stosować. Jeśli w jednej z sygnatur funkcji używane są opcje `width` i `height`, a w innej `w` i `h`, użytkownicy będą musieli ciągle zaglądać do dokumentacji i sprawdzać, gdzie należy stosować poszczególne nazwy. Podobnie jeśli w klasie `Widget` znajdują się metody do ustawiania właściwości, upewnij się, że wykorzystywane są w nich te same konwencje. Nie ma powodu, aby w jednej klasie używać metody `setWidth`, a w innej wykonywać identyczne operacje w metodzie `width`.

Każda dobra biblioteka wymaga kompletnej dokumentacji, jednak w najlepszych bibliotekach dokumentacja jest tylko kołem zapasowym. Gdy użytkownicy przyzwyczajają się do stosowanych w bibliotece konwencji, powinni móc wykonywać typowe zadania bez konieczności zaglądania do niej. Spójne konwencje pomagają nawet użytkownikom odgadywać, jakie właściwości i metody są dostępne, lub sprawdzać je w konsoli i ustalać ich działanie na podstawie nazw.

### Co warto zapamiętać?

- Stosuj spójne konwencje dla nazw zmiennych i sygnatur funkcji.
- Nie odchodź od konwencji, które użytkownicy prawdopodobnie znają z innych części platformy.



## Sposób 54. Traktuj wartość undefined jak brak wartości

Wartość `undefined` jest wyjątkowa. Gdy JavaScript nie może zwrócić konkretnej wartości, wyświetla `undefined`. Zmienne, do których nic nie przypisano, początkowo mają wartość `undefined`.

```
var x;  
x; // undefined
```

Próba dostępu do nieistniejących właściwości obiektów też prowadzi do zwrócenia wartości `undefined`.

```
var obj = {};  
obj.x; // undefined
```

Zwrócenie sterowania bez ustawionej wartości lub wyjście poza koniec ciała funkcji również powoduje zwrócenie wartości `undefined`.

```
function f() {  
    return;  
}  
  
function g() { }  
  
f(); // undefined  
g(); // undefined
```

Parametry funkcji, dla których nie podano argumentów, też mają wartość `undefined`.

```
function f(x) {  
    return x;  
}  
  
f(); // undefined
```

W każdej z tych sytuacji wartość `undefined` oznacza, że operacja nie zwróciła konkretnej wartości. Oczywiście jest coś paradoksalnego w wartości oznaczającej „brak wartości”. Jednak każda operacja musi zwrócić coś, dlatego JavaScript wykorzystuje wartość `undefined` do wypełniania luk.

Traktowanie wartości `undefined` jak braku konkretnej wartości to konwencja używana w języku. Stosowanie tej wartości w innych celach to ryzykowne podejście. Na przykład biblioteka elementów interfejsu użytkownika może udostępniać metodę `highlight` służącą do zmiany koloru tła elementu.

```
element.highlight(); // Zastosowanie koloru domyślnego  
element.highlight("yellow"); // Zastosowanie niestandardowego koloru
```

W jaki sposób zażądać ustawienia losowo wybranego koloru? Można wykorzystać w tym celu `undefined` jako specjalną wartość.

```
element.highlight(undefined); // Ustawienie losowo wybranego koloru
```

Jest to jednak niezgodne ze standardowym znaczeniem wartości `undefined`. Dlatego przy pobieraniu danych z zewnętrznego źródła (zwłaszcza takiego, w którym wartość może być niedostępna) mogą wystąpić problemy. Załóżmy, że program wykorzystuje obiekt z konfiguracją, w którym opcjonalnie określony jest kolor.

```
var config = JSON.parse(preferences);  
// ...  
element.highlight(config.highlightColor); // Możliwe, że ustawiony zostanie losowy kolor
```

Jeśli w konfiguracji kolor nie jest podany, programista prawdopodobnie spodziewa się, że ustawiony zostanie kolor domyślny (tak samo jak w sytuacji, gdy nie jest podana żadna wartość). Jednak ponieważ wartość `undefined` jest wykorzystywana w niestandardowym celu, brak koloru w konfiguracji prowadzi do ustawienia losowej barwy. Lepszym rozwiązaniem jest użycie w interfejsie API specjalnej nazwy do ustawiania losowych kolorów.

```
element.highlight("random");
```

Jednak czasem w interfejsie API nie da się wybrać specjalnego łańcucha znaków różnego od normalnego zestawu wartości przyjmowanych przez funkcję. Wtedy można wykorzystać inne wartości specjalne, różne od `undefined`. Są to na przykład `null` i `true`. Jednak zwykle kod jest wtedy niezrozumiały.

```
element.highlight(null);
```

Jeśli użytkownik kodu nie zna danej biblioteki na pamięć, trudno będzie mu zrozumieć tę instrukcję. Może nawet pomyśleć, że powoduje ona usunięcie wyróżnienia. Bardziej jednoznaczne i opisowe rozwiązanie to użycie w obiekcie właściwości `random` (więcej informacji o obiektach z opcjami zawiera sposób 55.).

```
element.highlight({ random: true });
```

Innym miejscem, w którym należy zwracać uwagę na wartość `undefined`, jest implementacja opcjonalnych argumentów. Obiekt `arguments` (zobacz sposób 51.) teoretycznie umożliwia wykrycie, czy argument został przekazany. Jednak w praktyce lepiej jest sprawdzać wartość `undefined`. Interfejs API jest wtedy stabilniejszy. Na przykład serwer WWW może przyjmować opcjonalną nazwę hosta.

```
var s1 = new Server(80, "example.com");  
var s2 = new Server(80); // Domyślnie używana jest wartość "localhost"
```

W konstruktorze klasy `Server` można sprawdzać wartość właściwości `arguments.length`.

```
function Server(port, hostname) {  
    if (arguments.length < 2) {  
        hostname = "localhost";  
    }  
}
```

```
    hostname = String(hostname);  
    // ...  
}
```

Prowadzi to jednak do podobnych problemów jak w pokazanej wcześniej metodzie `element.highlight`. Jeśli program bezpośrednio ustawi argument w wyniku zażądania wartości z innego źródła (na przykład z obiektu z konfiguracją), uzyskaną wartością może być `undefined`.

```
var s3 = new Server(80, config.hostname);
```

Jeśli obiekt `config` nie zawiera ustawionej właściwości `hostname`, program powinien używać wartości domyślnej `"localhost"`. Jednak w przedstawionej implementacji nazwa hosta jest ustawiana wtedy na `"undefined"`. Dlatego lepiej jest sprawdzać wartość `undefined`. Pojawia się ona, gdy argument nie jest podany lub gdy wyrażenie przekazane jako argument ma wartość `undefined`.

```
function Server(port, hostname) {  
    if (hostname === undefined) {  
        hostname = "localhost";  
    }  
    hostname = String(hostname);  
    // ...  
}
```

Dobłą alternatywą jest sprawdzanie, czy parametr `hostname` ma wartość oznaczającą prawdę (zobacz sposób 3.).

```
function Server(port, hostname) {  
    hostname = String(hostname || "localhost");  
    // ...  
}
```

W tej wersji używany jest operator logiczny LUB (`||`). Zwraca on pierwszy argument, jeśli ma on wartość oznaczającą prawdę; w przeciwnym razie zwracany jest drugi argument. Tak więc jeżeli parametr `hostname` ma wartość `undefined` lub jest pustym łańcuchem znaków, wyrażenie `(hostname || "localhost")` zwraca wartość `"localhost"`. Użyty test sprawdza nie tylko wartość `undefined`. Wszystkie wartości oznaczające fałsz są traktowane w taki sam sposób jak `undefined`. W klasie `Server` jest to akceptowalne, ponieważ pusty łańcuch znaków nie jest poprawną nazwą hosta. Dlatego jeśli odpowiada Ci mniej ścisły interfejs API, który przekształca wszystkie wartości oznaczające fałsz na wartość domyślną, sprawdzanie prawdziwości jest zwięzłym sposobem na zaimplementowanie ustawiania wartości domyślnej parametru.

Uważaj jednak — sprawdzanie prawdziwości nie zawsze jest bezpieczne. Jeżeli pusty łańcuch znaków jest poprawnym argumentem funkcji, test prawdziwości i tak spowoduje zastąpienie pustego łańcucha znaków wartością domyślną. Podobnie w funkcjach przyjmujących liczby nie należy stosować testu prawdziwości, jeśli wartość `0` (lub, co zdarza się rzadziej, `NaN`) jest dopuszczalna.

Na przykład funkcja do tworzenia elementu interfejsu użytkownika może dopuszczać tworzenie elementów o szerokości lub wysokości równej 0 i ustawiać inne wartości domyślne.

```
var c1 = new Element(0, 0); // Szerokość: 0, wysokość: 0
var c2 = new Element();    // Szerokość: 320, wysokość: 240
```

Implementacja, w której sprawdzana jest prawdziwość, zadziała nieprawidłowo.

```
function Element(width, height) {
  this.width = width || 320; // Niewłaściwy test
  this.height = height || 240; // Niewłaściwy test
  // ...
}
```

```
var c1 = new Element(0,0);
```

```
c1.width; // 320
c1.height; // 240
```

Zamiast tego trzeba zastosować dłuższy test wartości undefined.

```
function Element(width, height) {
  this.width = width === undefined ? 320 : width;
  this.height = height === undefined ? 240 : height;
  // ...
}
```

```
var c1 = new Element(0, 0);
```

```
c1.width; // 0
c1.height; // 0
```

```
var c2 = new Element();
```

```
c2.width; // 320
c2.height; // 240
```

## Co warto zapamiętać?

- Unikaj stosowania wartości undefined do reprezentowania czegoś innego niż brak konkretnej wartości.
- Do reprezentowania opcji aplikacji stosuj opisowe łańcuchy znaków lub obiekty z wartościami logicznymi zamiast wartości undefined lub null.
- Przy ustawianiu wartości domyślnych parametrów sprawdzaj wartość undefined zamiast właściwości arguments.length.
- Nigdy nie stosuj testów prawdziwości przy ustawianiu wartości domyślnej parametrów, dla których dozwolone są wartości 0, NaN lub puste łańcuchy znaków.

## Sposób 55. Stosuj obiekty z opcjami do przekazywania argumentów za pomocą słów kluczowych

Spójne określanie kolejności argumentów (zgodnie z sugestią ze sposobu 53.) jest ważne, ponieważ pomaga programistom zapamiętać znaczenie poszczególnych argumentów funkcji. Jednak to podejście sprawdza się tylko w niektórych sytuacjach. Trudno je stosować, gdy liczba argumentów jest duża. Spróbuj zrozumieć znaczenie poniższego wywołania funkcji.

```
var alert = new Alert(100, 75, 300, 200,  
    "Error", message,  
    "blue", "white", "black",  
    "error", true);
```

Z pewnością zetknąłeś się już z takimi interfejsami API. Często jest to efekt **do-dawania argumentów**. Polega to na tym, że funkcja początkowo jest prosta, jednak z czasem, wraz z rozwijaniem biblioteki, w sygnaturze pojawiają się coraz to nowe argumenty.

Na szczęście JavaScript udostępnia prosty idiom przydatny dla funkcji z długimi sygnaturami. Jest to **obiekt z opcjami**. Taki obiekt to jeden argument, w którym dane są podawane za pomocą nazwanych właściwości. Składnia literałów obiektowych sprawia, że pisanie i czytanie kodu opartego na tym idiomie jest bardzo łatwe.

```
var alert = new Alert({  
    x: 100, y: 75,  
    width: 300, height: 200,  
    title: "Error", message: message,  
    titleColor: "blue", bgColor: "white", textColor: "black",  
    icon: "error", modal: true  
});
```

Ten interfejs API jest dłuższy, ale dużo czytelniejszy. Każdy argument *sam się dokumentuje*. Nie są potrzebne komentarze wyjaśniające przeznaczenie argumentów, ponieważ nazwy właściwości są doskonałym wyjaśnieniem. Jest to przydatne zwłaszcza w przypadku parametrów logicznych, takich jak `modal`. Osoba czytająca wywołanie `new Alert` może zrozumieć przeznaczenie tekstowych argumentów na podstawie ich treści, jednak same wartości `true` lub `false` nie są pod tym względem przydatne.

Inną zaletą obiektów z opcjami jest to, że dowolny argument może być opcjonalny. Jednostka wywołująca może więc przekazać dowolny podzbiór argumentów opcjonalnych. W przypadku zwykłych argumentów (*opartych na pozycji*, ponieważ są określane nie według nazw, a na podstawie pozycji na liście argumentów) argumenty opcjonalne mogą prowadzić do niejednoznaczności. Jeśli pozycja i wielkość obiektu typu `Alert` są opcjonalne, nie jest oczywiste, jak należy zinterpretować poniższe wywołanie.

```
var alert = new Alert(app,
    150, 150,
    "Error", message,
    "blue", "white", "black",
    "error", true);
```

Czy dwie pierwsze liczby reprezentują argumenty  $x$  i  $y$ , czy  $width$  i  $height$ ?  
Gdy zastosujesz obiekt z opcjami, będzie to oczywiste.

```
var alert = new Alert({
    parent: app,
    width: 150, height: 100,
    title: "Error", message: message,
    titleColor: "blue", bgColor: "white", textColor: "black",
    icon: "error", modal: true
});
```

Przyjęło się, że obiekty z opcjami zawierają wyłącznie argumenty opcjonalne, dlatego można nawet zupełnie pominąć taki obiekt.

```
var alert = new Alert(); // Dla wszystkich parametrów używane są wartości domyślne
```

Gdy występują jeden lub dwa argumenty wymagane, lepiej jest podawać je niezależnie od obiektu z opcjami.

```
var alert = new Alert(app, message, {
    width: 150, height: 100,
    title: "Error",
    titleColor: "blue", bgColor: "white", textColor: "black",
    icon: "error", modal: true
});
```

Implementowanie funkcji, która przyjmuje obiekty z opcjami, wymaga jednak dodatkowej pracy. Oto kompletna implementacja takiej funkcji.

```
function Alert(parent, message, opts) {
    opts = opts || {}; // Domyślnie obiekt z opcjami jest pusty
    this.width = opts.width === undefined ? 320 : opts.width;
    this.height = opts.height === undefined
        ? 240
        : opts.height;
    this.x = opts.x === undefined
        ? (parent.width / 2) - (this.width / 2)
        : opts.x;
    this.y = opts.y === undefined
        ? (parent.height / 2) - (this.height / 2)
        : opts.y;
    this.title = opts.title || "Alert";
    this.titleColor = opts.titleColor || "gray";
    this.bgColor = opts.bgColor || "white";
    this.textColor = opts.textColor || "black";
    this.icon = opts.icon || "info";
    this.modal = !!opts.modal;
    this.message = message;
}
```

Kod rozpoczyna się od utworzenia domyślnego pustego obiektu z opcjami przy użyciu operatora `||` (zobacz sposób 54.). Dla argumentów liczbowych sprawdzana jest wartość `undefined` (zgodnie z radą ze sposobu 54.), ponieważ wartość `0` jest dopuszczalna, ale nie jest domyślna. Dla parametrów tekstowych używany jest operator logiczny LUB, ponieważ pusty łańcuch znaków nie jest tu uznawany za prawidłowy, dlatego należy go zastąpić wartością domyślną. Argument odpowiadający parametrowi `modal` jest przekształcany na wartość logiczną za pomocą podwójnej negacji (`!!`).

Ten kod jest dłuższy niż dla argumentów podawanych na podstawie pozycji. Jednak tworząc bibliotekę, warto pogodzić się z dodatkowymi kosztami, jeśli ma to ułatwić pracę użytkownikom. Możesz jednak uprościć sobie zadanie za pomocą przydatnej abstrakcji — funkcji *rozszerzającej* (lub *scalającej*) obiekt. Wiele bibliotek i platform JavaScriptu udostępnia funkcję `extend`. Przyjmuje ona obiekt *docelowy* i obiekt *źródłowy* oraz kopiuje właściwości z obiektu źródłowego do docelowego. Jednym z najczęstszych zastosowań tego mechanizmu jest scalanie wartości domyślnych i wartości podanych przez użytkownika w obiektach z opcjami. Po zastosowaniu funkcji `extend` funkcja `Alert` staje się bardziej przejrzysta.

```
function Alert(parent, message, opts) {
    opts = extend({
        width: 320,
        height: 240
    });
    opts = extend({
        x: (parent.width / 2) - (opts.width / 2),
        y: (parent.height / 2) - (opts.height / 2),
        title: "Alert",
        titleColor: "gray",
        bgColor: "white",
        textColor: "black",
        icon: "info",
        modal: false
    }, opts);
    this.width = opts.width;
    this.height = opts.height;
    this.x = opts.x;
    this.y = opts.y;
    this.title = opts.title;
    this.titleColor = opts.titleColor;
    this.bgColor = opts.bgColor;
    this.textColor = opts.textColor;
    this.icon = opts.icon;
    this.modal = opts.modal;
}
```

To pozwala uniknąć wielokrotnego pisania kodu sprawdzającego obecność poszczególnych argumentów. Zwróć uwagę, że występują tu dwa wywołania funkcji `extend`, ponieważ domyślne wartości parametrów `x` i `y` wymagają wcześniejszego wyznaczenia wartości parametrów `width` i `height`.

Jeśli celem jest tylko skopiowanie opcji do obiektu `this`, kod można uprościć jeszcze bardziej.

```
function Alert(parent, message, opts) {
  opts = extend({
    width: 320,
    height: 240
  });
  opts = extend({
    x: (parent.width / 2) - (opts.width / 2),
    y: (parent.height / 2) - (opts.height / 2),
    title: "Alert",
    titleColor: "gray",
    bgColor: "white",
    textColor: "black",
    icon: "info",
    modal: false
  }, opts);
  extend(this, opts);
}
```

Poszczególne platformy udostępniają różne wersje funkcji `extend`. Zwykle pobiera ona właściwości obiektu źródłowego i kopiuje je do docelowego, jeśli wartość danej właściwości jest różna od `undefined`.

```
function extend(target, source) {
  if (source) {
    for (var key in source) {
      var val = source[key];
      if (typeof val !== "undefined") {
        target[key] = val;
      }
    }
  }
  return target;
}
```

Zwróć uwagę na drobne różnice między pierwotną wersją funkcji `Alert` a implementacją wykorzystującą funkcję `extend`. W pierwszej wersji instrukcje warunkowe nie obliczają wartości domyślnych, jeśli nie są one potrzebne. Gdy wyznaczanie wartości domyślnych nie powoduje efektów ubocznych (takich jak modyfikowanie interfejsu użytkownika lub przesyłanie żądań sieciowych), nie stanowi to problemu. Inna różnica dotyczy kodu sprawdzającego dostępność wartości. W pierwszej wersji dla argumentów tekstowych pusty łańcuch znaków jest traktowany tak samo jak wartość `undefined`. Jednak spójniejszym rozwiązaniem jest traktowanie tylko wartości `undefined` jako braku argumentu. Zastosowanie operatora `||` to wygodniejszy, ale mniej spójny sposób ustawiania wartości domyślnych parametrów. W trakcie projektowania biblioteki warto dbać o spójność, ponieważ interfejs API jest wtedy bardziej przewidywalny dla jego użytkowników.



## Co warto zapamiętać?

- Obiekty z opcjami sprawiają, że interfejsy API są czytelniejsze i łatwiejsze do zapamiętania.
- Wszystkie argumenty podawane w obiekcie z opcjami powinny być opcjonalne.
- Używaj funkcji narzędziowej `extend` do pobierania wartości z obiektów z opcjami.

## Sposób 56. Unikaj niepotrzebnego przechowywania stanu

Interfejsy API opisuje się czasem jako **stanowe** lub **bezstanowe**. Bezstanowy interfejs API udostępnia funkcje i metody, których działanie zależy tylko od danych wejściowych, a nie od zmian w stanie programu. Metody łańcuchów znaków są bezstanowe. Zawartości łańcucha znaków nie można modyfikować, dlatego metody zależą tylko od treści łańcucha i przekazanych argumentów. Niezależnie od tego, co dzieje się w programie, wyrażenie `"foo".toUpperCase()` zawsze zwraca wartość `"FOO"`. Natomiast metody obiektów typu `Date` są stanowe. Wywołanie metody `toString` dla tego samego obiektu typu `Date` może zwracać różne wyniki w zależności od tego, czy właściwości obiektu zostały zmodyfikowane przy użyciu różnych metod z rodziny `set`, czy nie.

Choć stan czasem jest niezbędny, bezstanowe interfejsy API są zwykle łatwiejsze do opanowania i w użyciu, w większym stopniu samodokumentujące i mniej narażone na błędy. Znanym stanowym interfejsem API jest używana przy tworzeniu stron internetowych biblioteka `Canvas`, która udostępnia elementy interfejsu użytkownika z metodami służącymi do wyświetlania figur geometrycznych i rysunków w oknie. Program może wyświetlić tekst w oknie za pomocą metody `fillText`:

```
c.fillText("Witaj, świecie!", 75, 25);
```

Ta metoda przyjmuje wyświetlany łańcuch znaków i jego pozycję w oknie. Nie określa jednak innych atrybutów wyświetlanego tekstu (na przykład koloru, stopnia przezroczystości lub stylu). Te atrybuty są określane osobno. Wymaga to zmiany wewnętrznego stanu okna.

```
c.fillStyle = "blue";  
c.font = "24pt serif";  
c.textAlign = "center";  
c.fillText("Witaj, świecie!", 75, 25);
```

Wersja tego interfejsu API w mniejszym stopniu oparta na stanie wygląda tak:

```
c.fillText("Witaj, świecie!", 75, 25, {
    fillStyle: "blue",
    font: "24pt serif",
    textAlign: "center"
});
```

Dlaczego ta druga wersja jest zalecana? Przede wszystkim jest mniej narażona na błędy. Stanowy interfejs API wymaga zmiany wewnętrznego stanu okna w celu wykonania niestandardowych operacji. Wtedy jedna operacja wyświetlania wpływa na inne, nawet jeśli nie mają one ze sobą nic wspólnego. Na przykład domyślne wypełnienie jest czarne. Jednak wartość domyślna jest używana tylko wtedy, jeśli nikt inny jej nie zmienił. Jeśli chcesz wykorzystać domyślny kolor po jego wcześniejszej zmianie, musisz ustawić go bezpośrednio.

```
c.fillText("Tekst 1.", 0, 0); // Kolor domyślny
c.fillStyle = "blue";
c.fillText("Tekst 2.", 0, 30); // Niebieski
c.fillStyle = "black";
c.fillText("Tekst 3.", 0, 60); // Ponownie czarny
```

Porównaj to z wersją opartą na bezstanowym interfejsie API, w którym można automatycznie wykorzystać wartości domyślne.

```
c.fillText("Tekst 1.", 0, 0); // Kolor domyślny
c.fillText("Tekst 2.", 0, 30, { fillStyle: "blue"}); // Niebieski
c.fillText("Tekst 3.", 0, 60); // Kolor domyślny
```

Zauważ, że instrukcje są teraz bardziej czytelne. Aby zrozumieć, jak działa każde wywołanie metody `fillText`, nie musisz znać wszystkich wcześniejszych modyfikacji stanu. Okno może nawet być modyfikowane w zupełnie innej części programu. W wersji stanowej łatwo prowadzi to do błędów, gdy fragment kodu z innego miejsca aplikacji zmienia stan okna.

```
c.fillStyle = "blue";
drawMyImage(c); // Czy funkcja drawMyImage modyfikuje stan obiektu c?
c.fillText("Witaj, świecie!", 75, 25);
```

Aby zrozumieć, co dzieje się w ostatnim wierszu, trzeba wiedzieć, jakie modyfikacje funkcja `drawMyImage` wprowadza w oknie. Bezstanowy interfejs API pozwala tworzyć bardziej modułowy kod i uniknąć błędów wynikających z nieoczekiwanych interakcji między różnymi fragmentami programu, a jednocześnie poprawia czytelność kodu.

Stanowe interfejsy API są często trudniejsze do opanowania. Na podstawie dokumentacji metody `fillText` trudno jest ustalić, które aspekty stanu okna wpływają na proces rysowania. Nawet jeśli niektóre rzeczy łatwo jest odgadnąć, początkujący użytkownik będzie miał trudności z ustaleniem, czy poprawnie zainicjował cały potrzebny stan. Oczywiście można udostępnić kompletną listę wymaganych ustawień w dokumentacji metody `fillText`.

Gdy niezbędny jest stanowy interfejs API, trzeba starannie udokumentować wszelkie przypadki zależności od stanu. Jednak bezstanowy interfejs API eliminuje niejawne zależności, dlatego nie wymagają one dodatkowej dokumentacji.

Inną zaletą bezstanowych interfejsów API jest spójność. Stanowe interfejsy API zwykle wymagają wielu dodatkowych instrukcji ustawiających wewnętrzny stan obiektu przed wywołaniem jego metod. Zastanów się nad parserem popularnego formatu plików konfiguracyjnych — *.ini*. Prosty plik *.ini* może zawierać następujące dane:

```
[Host]
address=172.0.0.1
name=localhost
[Connections]
timeout=10000
```

Możliwym interfejsem API dla danych tego rodzaju jest metoda `setSection` pozwalająca na wybór sekcji przed sprawdzeniem parametrów konfiguracyjnych za pomocą metody `get`.

```
var ini = INI.parse(src);

ini.setSection("Host");
var addr = ini.get("address");
var hostname = ini.get("name");
ini.setSection("Connection");
var timeout = ini.get("timeout");
var server = new Server(addr, hostname, timeout);
```

Jednak bezstanowy interfejs API nie wymaga tworzenia dodatkowych zmiennych (takich jak `addr` lub `hostname`) do przechowywania pobranych danych przed aktualizacją sekcji.

```
var ini = INI.parse(src);
var server = new Server(ini.Host.address,
                        ini.Host.name,
                        ini.Connection.timeout);
```

Zauważ, że dzięki jawnemu określeniu sekcji można przedstawić obiekt `ini` i poszczególne sekcje za pomocą słownika. W ten sposób interfejs API stał się jeszcze prostszy. Więcej o obiektach reprezentujących słowniki dowiesz się z rozdziału 5.

## Co warto zapamiętać?

- Jeśli to możliwe, stosuj bezstanowe interfejsy API.
- Gdy używasz stanowego interfejsu API, udokumentuj stan, od którego zależy każda operacja.

## Sposób 57. Określaj typy na podstawie struktury, aby tworzyć elastyczne interfejsy

Wyobraź sobie bibliotekę do tworzenia serwisów **wiki** — witryn zawierających informacje, które użytkownicy mogą interaktywnie tworzyć, usuwać i modyfikować. Wiele serwisów wiki obsługuje prosty, tekstowy język znaczników służący do tworzenia zawartości witryny. Takie języki znaczników zwykle udostępniają podzbiór mechanizmów HTML-a, ale mają prostszy i łatwiejszy do zrozumienia format. Na przykład tekst można sformatować przez umieszczenie go między symbolami gwiazdki (w celu pogrubienia), podkreślenia (w celu dodania podkreślenia) lub ukośników (w celu dodania kursywy). Użytkownicy mogą więc wpisać tekst o następującej postaci:

To zdanie zawiera *\*pogrubione\** słowo.

To zdanie zawiera *\_podkreślone\_* słowo.

To zdanie zawiera słowo zapisane */kursywą/*.

W witrynie tekst będzie wyświetlany czytelnikom serwisu wiki w następujący sposób:

To zdanie zawiera **pogrubione** słowo.

To zdanie zawiera podkreślone słowo.

To zdanie zawiera słowo zapisane *kursywą*.

Elastyczna biblioteka do tworzenia serwisów wiki może zapewniać autorom aplikacji wybór języków znaczników, ponieważ przez lata pojawiło się wiele popularnych odmian.

Aby to podejście zadziało, trzeba oddzielić mechanizm pobierania przygotowanego przez użytkownika tekstu w języku znaczników od pozostałych funkcji serwisu wiki (takich jak zarządzanie kontem, historia poprawek i przechowywanie informacji). Pozostałe części aplikacji powinny komunikować się z mechanizmem pobierania poprzez *interfejs* z dobrze udokumentowanym zbiorem właściwości i metod. Dzięki programowaniu zgodnie z udokumentowanym interfejsem API i ignorowaniu szczegółów implementacji reszta aplikacji może działać prawidłowo niezależnie od źródłowego formatu wybranego w aplikacji.

Przyjrzyjmy się bliżej temu, jaki rodzaj interfejsu jest potrzebny do pobierania zawartości serwisu wiki. Biblioteka musi mieć możliwość pobierania metadanych (na przykład tytułu i autora strony) i formatowania treści stron jako kodu w języku HTML, aby wyświetlać ją czytelnikom serwisu wiki. Każdą stronę serwisu można reprezentować jako obiekt, który zapewnia dostęp do danych za pomocą metod takich jak `getTitle`, `getAuthor` i `toHTML`.

Biblioteka potrzebuje też sposobu na utworzenie aplikacji z niestandardowym mechanizmem formatowania stron serwisu wiki i wbudowanymi mechanizma-

mi formatowania dla popularnych języków znaczników. Autor aplikacji może na przykład chcieć zastosować format MediaWiki (używany w Wikipedii).

```
var app = new Wiki(Wiki.formats.MEDIAWIKI);
```

Biblioteka przechowuje funkcję formatującą wewnętrznie w egzemplarzu obiektu typu Wiki.

```
function Wiki(format) {  
    this.format = format;  
}
```

Gdy czytelnik chce wyświetlić stronę, aplikacja pobiera jej kod źródłowy i wyświetla stronę HTML-ową za pomocą wewnętrznego mechanizmu formatującego.

```
Wiki.prototype.displayPage = function(source) {  
    var page = this.format(source);  
    var title = page.getTitle();  
    var author = page.getAuthor();  
    var output = page.toHTML();  
    // ...  
};
```

Jak zaimplementować mechanizm formatujący, taki jak Wiki.formats.MEDIAWIKI? Programiści przyzwyczajeni do używania klas prawdopodobnie utworzą klasę bazową Page, która reprezentuje treść przygotowaną przez użytkownika. Obsługa poszczególnych formatów znajdzie się wtedy w klasach pochodnych od klasy Page. Format MediaWiki można zaimplementować w klasie MWPage (pochodnej od klasy Page), a funkcję MEDIAWIKI napisać jako funkcję fabryczną zwracającą egzemplarz typu MWPage.

```
function MWPage(source) {  
    Page.call(this, source); // Wywołanie konstruktora klasy bazowej  
    // ...  
}
```

```
// Klasa MWPage dziedziczy po klasie Page  
MWPage.prototype = Object.create(Page.prototype);
```

```
MWPage.prototype.getTitle = /* ... */;  
MWPage.prototype.getAuthor = /* ... */;  
MWPage.prototype.toHTML = /* ... */;
```

```
Wiki.formats.MEDIAWIKI = function(source) {  
    return new MWPage(source);  
};
```

Więcej informacji o implementowaniu hierarchii klas za pomocą konstruktorów i prototypów zawiera rozdział 4. Jednak do czego tak naprawdę potrzebna jest tu klasa bazowa Page? Ponieważ klasa MWPage musi mieć własną implementację metod niezbędnych w aplikacji obsługującej serwis wiki (getTitle, getAuthor i toHTML), nie dziedziczy żadnego przydatnego kodu. Ponadto dla

metody `displayPage` hierarchia dziedziczenia nie jest istotna. Ta metoda potrzebuje tylko innych odpowiednich metod. Dlatego w implementacjach formatów można napisać wymienione metody w dowolny sposób.

Choć wiele języków obiektowych zachęca do budowania struktury programów na podstawie klas i dziedziczenia, w JavaScriptcie jest inaczej. Często wystarczy zaimplementować interfejs (taki jak dla formatu MediaWiki) za pomocą prostego literału obiektowego.

```
Wiki.formats.MEDIAWIKI = function(source) {  
  // Pobieranie treści z kodu źródłowego  
  // ...  
  return {  
    getTitle: function() { /* ... */ },  
    getAuthor: function() { /* ... */ },  
    toHTML: function() { /* ... */ }  
  };  
};
```

Co więcej, dziedziczenie czasem powoduje więcej problemów, niż ich rozwiązuje. Staje się to oczywiste, gdy poszczególne pary formatów mają różne cechy wspólne. W takiej sytuacji utworzenie sensownej hierarchii dziedziczenia może okazać się niemożliwe. Załóżmy, że używane są trzy podane poniżej formaty.

Format A: *\*pogrubienie\**, [Oдноśnik], /kursywa/

Format B: **\*\*pogrubienie\*\***, [[Oдноśnik]], \*kursywa\*

Format C: **\*\*pogrubienie\*\***, [Oдноśnik], \*kursywa\*

Programista chce zaimplementować mechanizm do wykrywania znaków formatujących, ale nie potrafi utworzyć odpowiedniej hierarchii dla formatów A, B i C (zachęcamy do samodzielnej próby wykonania tego zadania). Właściwe rozwiązanie polega na zaimplementowaniu odrębnych funkcji do dopasowywania poszczególnych znaków formatujących (pojedynczych gwiazdek, podwójnych gwiazdek, nawiasów kwadratowych itd.) oraz łączeniu ich w poszczególnych formatach.

Zauważ, że po rezygnacji z klasy bazowej `Page` nie trzeba niczym jej zastępować. W tym obszarze stosowane w JavaScriptcie dynamiczne określanie typu pokazuje swoją wartość. Programista, który chce zaimplementować nowy format, nie musi go nigdzie rejestrować. Metoda `displayPage` zadziała dla dowolnego obiektu w JavaScriptcie, o ile obiekt ten ma właściwą strukturę — udostępnia metody `getTitle`, `getAuthor` i `getHTML` działające w oczekiwany sposób.

Tego rodzaju interfejs oparty jest na **określaniu typu na podstawie struktury** (ang. *duck typing*, czyli „kacze typowanie” — jeśli coś wygląda jak kaczką, pływa jak kaczką i kwacze jak kaczką, to pewnie jest kaczką). Dozwolony jest więc dowolny obiekt o oczekiwanej strukturze. Jest to elegancki wzorzec

programowania, szczególnie prosty w językach dynamicznych (takich jak JavaScript), ponieważ nie wymaga pisania żadnych specjalnych konstrukcji. Funkcja wywołująca metody danego obiektu działa dla dowolnego obiektu z odpowiednim interfejsem. Oczywiście należy opisać oczekiwania wobec interfejsu obiektu w dokumentacji interfejsu API. Dzięki temu autor implementacji będzie wiedział, które właściwości i metody są wymagane, a także jakiego działania oczekują od nich biblioteki i aplikacje.

Elastyczność zapewniana przez określanie typu na podstawie struktury jest też przydatna w zakresie testów jednostkowych. Biblioteka serwisu wiki prawdopodobnie będzie podłączana do obiektu serwera HTTP obsługującego mechanizmy sieciowe serwisu. W celu przetestowania sekwencji interakcji z serwisem wiki bez łączenia się z siecią można przygotować **atrapę**, która pozornie działa jak aktywny serwer HTTP, ale w rzeczywistości wykonuje operacje według skryptu, bez łączenia się z siecią. W ten sposób można uzyskać powtarzalne interakcje z symulowanym serwerem i nie trzeba polegać na nieprzewidywalnej pracy sieci. Pozwala to testować działanie komponentów komunikujących się z serwerem.

### Co warto zapamiętać?

- Stosuj określanie typu na podstawie struktury (inna nazwa to *duck typing*), aby uzyskać obiekty o elastycznych interfejsach.
- Unikaj dziedziczenia w sytuacjach, gdy interfejsy oparte na strukturze są prostsze i bardziej elastyczne.
- Na potrzeby testów jednostkowych stosuj atrapy, czyli zastępcze implementacje interfejsu zapewniające powtarzalne działania.

## Sposób 58. Różnice między tablicami a obiektami podobnymi do tablic

Przyjrzyj się dwóm różnym interfejsom API klas. Pierwszy jest przeznaczony dla **wektorów bitowych**, czyli dla uporządkowanych kolekcji bitów.

```
var bits = new BitVector();

bits.enable(4);
bits.enable([1, 3, 8, 17]);

bits.bitAt(4); // 1
bits.bitAt(8); // 1
bits.bitAt(9); // 0
```

Zauważ, że metoda `enable` jest *przeciążona*. Można przekazać do niej indeks lub tablicę indeksów.

Drugi interfejs API dotyczy **zbiorów łańcuchów znaków**, czyli nieuporządkowanych kolekcji łańcuchów znaków.

```
var set = new StringSet();

set.add("Hamlet");
set.add(["Rosencrantz", "Guildenstern"]);
set.add({ "Ophelia": 1, "Polonius": 1, "Horatio": 1 });

set.contains("Polonius");    // true
set.contains("Guildenstern"); // true
set.contains("Falstaff");    // false
```

Metoda `add` (podobnie jak metoda `enable` wektorów bitowych) jest przeciążona. Oprócz łańcuchów znaków i tablic łańcuchów znaków przyjmuje też słowniki.

W implementacji metody `BitVector.prototype.enable` można uniknąć sprawdzania, czy dany obiekt jest tablicą. W tym celu najpierw należy sprawdzać drugą możliwość.

```
BitVector.prototype.enable = function(x) {
  if (typeof x === "number") {
    this.enableBit(x);
  } else { // Zakładamy, że x to obiekt podobny do tablicy
    for (var i = 0, n = x.length; i < n; i++) {
      this.enableBit(x[i]);
    }
  }
};
```

To było proste. A co z metodą `StringSet.prototype.add`? Trzeba w niej odróżnić tablice od obiektów. Jednak to rozróżnienie nie ma sensu — w JavaScriptcie tablice są obiektami. W rzeczywistości trzeba oddzielić obiekty reprezentujące tablice od innych obiektów.

To rozróżnienie jest niezgodne z występującymi w JavaScriptcie i zapewniającymi dużą swobodę obiektami podobnymi do tablicy (zobacz sposób 51.). Dowolny obiekt można traktować jak tablicę, pod warunkiem że udostępnia odpowiedni interfejs. Nie istnieje prosty sposób na przetestowanie obiektu i ustalenie, czy ma mieć określony interfejs. Można zgadywać, że jeśli obiekt udostępnia właściwość `length`, ma być używany jak tablica, nie ma jednak takiej gwarancji. Możliwe przecież, że używany jest słownik z kluczem `"length"`.

```
dimensions.add({
  "length": 1, // Czy to oznacza, że obiekt jest podobny do tablicy?
  "height": 1,
  "width": 1
});
```

Stosowanie nieprecyzyjnych heurystyk do badania interfejsu to prosta droga do nieporozumień i błędów. Zgadywanie, czy obiekt ma strukturę określonego typu, nazywane jest czasem *duck testing* (przez analogię do określenia *duck*



*typing*; zobacz sposób 57.). Nie należy stosować tej techniki. Ponieważ obiekty nie mają jawnie zapisanych informacji określających implementowany strukturalnie typ, nie istnieje niezawodna, programowa metoda na ustalenie tych danych.

Uwzględnianie przy przeciążaniu dwóch typów oznacza, że musi istnieć sposób na ich odróżnienie. Nie da się jednak stwierdzić, czy w danym obiekcie zaimplementowano określony interfejs oparty na strukturze. To prowadzi do następującej reguły:

*W interfejsach API nigdy nie należy przeciążać opartych na strukturze typów przy użyciu innych pokrywających się typów.*

W klasie `StringSet` rozwiązaniem jest rezygnacja z opartego na strukturze interfejsu obiektu podobnego do tablicy. Zamiast tego należy zbudować typ ze ściśle zdefiniowaną informacją o tym, że użytkownik chce używać danego obiektu jak tablicy. Oczywiście, choć niedoskonałym podejściem jest zastosowanie operatora `instanceof` do sprawdzania, czy obiekt dziedziczy po prototypie `Array.prototype`.

```
StringSet.prototype.add = function(x) {
  if (typeof x === "string") {
    this.addString(x);
  } else if (x instanceof Array) { // To zbyt restrykcyjne rozwiązanie
    x.forEach(function(s) {
      this.addString(s);
    }, this);
  } else {
    for (var key in x) {
      this.addString(key);
    }
  }
};
```

Wiadomo, że gdy obiekt jest egzemplarzem typu `Array`, działa jak tablica. Jednak tym razem okazuje się, że test jest zbyt *drobiazgowy*. W środowiskach, w których może istnieć wiele obiektów globalnych, czasem znajduje się wiele kopii konstruktora i prototypu standardowego typu `Array`. Jest tak na przykład w przeglądarce, w której każda ramka ma własną kopię biblioteki standardowej. Gdy wartości są przekazywane między ramkami, tablica z jednej z nich nie dziedziczy po prototypie `Array.prototype` z innej ramki.

Dlatego w standardzie ES5 wprowadzono funkcję `Array.isArray`. Sprawdza ona, czy dana wartość jest tablicą. Nie opiera się przy tym na dziedziczeniu prototypów. W języku standardu ECMAScript funkcja sprawdza, czy wartość wewnętrznej właściwości `[[Class]]` to `"Array"`. Gdy trzeba sprawdzić, czy dany obiekt jest tablicą, a nie tylko obiektem podobnym do tablicy, funkcja `Array.isArray` jest bardziej niezawodna niż instrukcja `instanceof`.

To pozwala utworzyć mniej podatną na pomyłki implementację metody `add`.

```
StringSet.prototype.add = function(x) {
  if (typeof x === "string") {
    this.addString(x);
  } else if (Array.isArray(x)) { // Wykrywa prawdziwe tablice
    x.forEach(function(s) {
      this.addString(s);
    }, this);
  } else {
    for (var key in x) {
      this.addString(key);
    }
  }
};
```

W środowiskach, które nie obsługują specyfikacji ES5, do sprawdzania, czy obiekt jest tablicą, można wykorzystać standardową metodę `Object.prototype.toString`.

```
var toString = Object.prototype.toString;

function isArray(x) {
  return toString.call(x) === "[object Array]";
}
```

Funkcja `Object.prototype.toString` wykorzystuje wewnętrzną właściwość `[[Class]]` obiektu do utworzenia wynikowego łańcucha znaków. Jest więc bardziej niezawodną niż operator `instanceof` metodą sprawdzania, czy obiekt to tablica.

Zauważ, że ta wersja metody `add` działa w inny sposób, co wpływa na użytkowników nowego interfejsu API. Wersja przeciążonego interfejsu API przeznaczona dla tablic nie akceptuje dowolnych obiektów podobnych do tablicy. Nie można na przykład przekazać obiektu `arguments` i oczekiwać, że zostanie potraktowany jak tablica.

```
function MyClass() {
  this.keys = new StringSet();
  // ...
}

MyClass.prototype.update = function() {
  this.keys.add(arguments); // Argument jest traktowany jak słownik
};
```

Właściwy sposób używania metody `add` polega na przekształceniu obiektu na prawdziwą tablicę przy użyciu idiomu opisanego w sposobie 51.

```
MyClass.prototype.update = function() {
  this.keys.add([].slice.call(arguments));
};
```

Jednostka wywołująca musi przeprowadzić taką konwersję za każdym razem, gdy chce przekazać obiekt podobny do tablicy do interfejsu API oczekującego prawdziwej tablicy. Dlatego trzeba udokumentować, jakie dwa typy ten in-

terfejs przyjmuje. W przedstawionych wcześniej przykładach metoda `enable` przyjmuje liczby i obiekty podobne do tablic, natomiast metoda `add` przyjmuje łańcuchy znaków, prawdziwe tablice i obiekty różne od tablic.

### Co warto zapamiętać?

- Nigdy nie przeciążaj typów określanych na podstawie struktury pokrywającymi się typami.
- Gdy przeciążasz typy oparte na strukturze przy użyciu innych typów, najpierw sprawdzaj inne typy.
- Gdy używasz innych typów obiektowych przy przeciążaniu, przyjmuj prawdziwe tablice zamiast obiektów podobnych do tablicy.
- Opisz w dokumentacji, czy interfejs API przyjmuje tylko prawdziwe tablice, czy także wartości podobne do tablicy.
- Do wykrywania prawdziwych tablic używaj metody `Array.isArray` ze standardu ES5.

### Sposób 59. Unikaj nadmiernej koercji

JavaScript jest znany ze swobodnego traktowania typów (zobacz sposób 3.). Wiele standardowych operatorów i bibliotek automatycznie przekształca argumenty na oczekiwany typ, zamiast zgłaszać wyjątki dotyczące nieoczekiwanych danych wejściowych. Jeśli nie zastosujesz dodatkowego kodu, wykorzystanie takich wbudowanych operatorów doprowadzi do odziedziczenia operacji powodujących koercję.

```
function square(x) {  
    return x * x;  
}
```

```
square("3"); // 9
```

Koercja oczywiście bywa wygodna. Jednak w sposobie 3. wyjaśniono, że może też prowadzić do problemów, ukrywania błędów oraz niespójnych i trudnych do zdiagnozowania zachowań programu.

Koercje są kłopotliwe, zwłaszcza gdy używa się przeciążonych funkcji, takich jak metoda `enable` klasy wektora bitowego opisana w sposobie 58. Ta metoda określa wykonywane działania na podstawie typu argumentów. Sygnatura byłaby mniej zrozumiała, gdyby metoda `enable` próbowała przekształcać argumenty na oczekiwany typ. Który typ należy wybrać? Jeśli będzie to liczba, przeciążanie przestanie działać.

```

BitVector.prototype.enable = function(x) {
  x = Number(x);
  if (typeof x === "number") { // Ten warunek zawsze jest spełniony
    this.enableBit(x);
  } else { // Poniższy kod nigdy nie jest wykonywany
    for (var i = 0, n = x.length; i < n; i++) {
      this.enableBit(x[i]);
    }
  }
};

```

Zgodnie z ogólną regułą warto unikać koercji argumentów, których typ służy do określania działania przeciążonej funkcji. Koercje utrudniają ustalenie, która wersja kodu zostanie wykonana. Spróbuj zrozumieć działanie poniższego wywołania.

```
bits.enable("100"); // Liczba czy obiekt podobny do tablicy?
```

To wywołanie metody `enable` jest niejednoznaczne. Autor może chcieć, aby argument został potraktowany jak liczba lub jak tablica wartości bitowych. Jednak konstruktor nie został zaprojektowany z myślą o łańcuchach znaków, dlatego nie da się tego stwierdzić. Prawdopodobnie autor nie rozumiał używanego interfejsu API. Aby doprecyzować interfejs API, można pozwolić na podawanie wyłącznie liczb i obiektów.

```

BitVector.prototype.enable = function(x) {
  if (typeof x === "number") {
    this.enableBit(x);
  } else if (typeof x === "object" && x) {
    for (var i = 0, n = x.length; i < n; i++) {
      this.enableBit(x[i]);
    }
  } else {
    throw new TypeError("Podaj liczbę lub obiekt podobny do tablicy");
  }
}

```

Ostatnia wersja metody `enable` to przykład zastosowania ostrożnego stylu nazywanego **programowaniem defensywnym**. Polega on na zabezpieczaniu się przed błędami za pomocą dodatkowych testów. Ochrona przed wszelkimi problemami jest niemożliwa. Można na przykład sprawdzać, czy `x` to obiekt i czy ma właściwość `length`, jednak nie zabezpiecza to przed przypadkowym użyciem obiektu typu `String`. JavaScript udostępnia tylko podstawowe mechanizmy sprawdzania takich informacji, na przykład operator `typeof`. Można jednak napisać funkcje narzędziowe i dokładniej zabezpieczyć sygnatury funkcji. Konstruktor klasy `BitVector` można na przykład zabezpieczyć jednym wstępnym testem.

```

function BitVector(x) {
  uint32.or(arrayLike).guard(x);
  // ...
}

```

Aby to podejście zadziałało, można zbudować bibliotekę narzędziową obiektów zabezpieczających o wspólnym prototypie zawierającym metodę `guard`.

```
var guard = {
  guard: function(x) {
    if (!this.test(x)) {
      throw new TypeError("Oczekiwany typ: " + this);
    }
  }
};
```

W każdym obiekcie zabezpieczającym należy zaimplementować metodę `test` i podać opis tekstowy używany w komunikatach o błędach.

```
var uint32 = Object.create(guard);

uint32.test = function(x) {
  return typeof x === "number" && x === (x >>> 0);
};

uint32.toString = function() {
  return "uint32";
};
```

W obiekcie zabezpieczającym `uint32` zastosowano sztuczkę z operatorami bitowymi z JavaScriptu, aby przekształcać dane na 32-bitowe liczby całkowite bez znaku. *Operator przesunięcia w prawo bez uwzględniania znaku* przekształca pierwszy argument na 32-bitową liczbę całkowitą bez znaku, a następnie przeprowadza przesunięcie bitowe (zobacz sposób 2.). Przesunięcie o zero bitów nie wpływa na wartość liczby. Oznacza to, że metoda `uint32.test` porównuje liczbę z wynikiem jej konwersji na 32-bitową liczbę całkowitą bez znaku.

Teraz zobacz, jak zaimplementować obiekt zabezpieczający `arrayLike`.

```
var arrayLike = Object.create(guard);

arrayLike.test = function(x) {
  return typeof x === "object" && x && uint32.test(x.length);
};

arrayLike.toString = function() {
  return "array-like object";
};
```

Zauważ, że można pójść o krok dalej w programowaniu defensywnym i upewniać się, czy obiekt podobny do tablicy ma właściwość `length`, której wartość to liczba całkowita bez znaku.

Jeszcze inna możliwość to zaimplementowanie metod łączonych w łańcuch (ang. *chaining method*; zobacz sposób 60.), takich jak `or`, jako metod prototypu.

```
guard.or = function(other) {
  var result = Object.create(guard);
  var self = this;
```

```
result.test = function(x) {
    return self.test(x) || other.test(x);
};

var description = this + " or " + other;
result.toString = function() {
    return description;
};

return result;
};
```

Ta metoda łączy obiekt zabezpieczający odbiorcy (powiązany z nazwą `this`) z drugim obiektem zabezpieczającym (parametrem `other`). W efekcie powstaje nowy obiekt zabezpieczający, którego metody `test` i `toString` łączą dwie metody początkowych obiektów. Zauważ, że zmienna lokalna `self` jest używana do zapisania referencji do obiektu `this` (zobacz sposoby 25. i 37.) używanej w metodzie `test` wynikowego obiektu zabezpieczającego.

Opisane testy pomagają szybciej wykrywać błędy, co znacznie ułatwia ich diagnozę. Mogą jednak zaśmieszać kod bazowy i wpływać na wydajność aplikacji. Dlatego gdy zastanawiasz się nad zastosowaniem programowania defensywnego, rozważ koszty (liczbę dodatkowych testów, które trzeba napisać i wykonywać) oraz zyski (liczbę wcześniej wykrytych błędów, co przekłada się na krótszy czas rozwijania aplikacji i debugowania).

### Co warto zapamiętać?

- Nie łącz koercji z przeciążaniem.
- Pomyśl o defensywnym zabezpieczaniu się przed nieoczekiwanymi danymi wejściowymi.

## Sposób 60. Obsługa łańcuchów metod

Wartość bezstanowych interfejsów API (zobacz sposób 56.) po części wynika z tego, że umożliwiają budowanie złożonych operacji z prostych elementów. Doskonałym przykładem jest tu metoda `replace` łańcuchów znaków. Ponieważ jej wynik to łańcuch znaków, można wielokrotnie powtarzać zastępowanie, wywołując metodę `replace` dla wyniku poprzedniego wywołania. Ten wzorzec często stosuje się do zastępowania znaków specjalnych w łańcuchu przed umieszczeniem go na stronie HTML.

```
function escapeBasicHTML(str) {
    return str.replace(/&/g, "&amp;")
               .replace(/</g, "&lt;")
               .replace(/>/g, "&gt;")
               .replace(/"/g, "&quot;")
               .replace(/'/g, "&apos;");
}
```

Pierwsze wywołanie metody `replace` zwraca łańcuch znaków, w którym wszystkie wystąpienia specjalnego znaku "&" są zastąpione HTML-ową sekwencją "&". Drugie wywołanie zastępuje wszystkie wystąpienia znaku "<" sekwencją "&lt;" itd. Tego rodzaju wielokrotne wywoływanie metod to **łączenie metod w łańcuch**. Nie trzeba pisać kodu w ten sposób, jednak to rozwiązanie jest bardziej zwarte niż zapisywanie każdego pośredniego wyniku w pośredniej zmiennej.

```
function escapeBasicHTML(str1) {
  var str2 = str1.replace(/&/g, "&amp;");
  var str3 = str2.replace(/</g, "&lt;");
  var str4 = str3.replace(/>/g, "&gt;");
  var str5 = str4.replace(/"/g, "&quot;");
  var str6 = str5.replace(/'/g, "&apos;");
  return str6;
}
```

Wyliminowanie zmiennych tymczasowych ułatwia czytelnikom kodu zrozumienie, że pośrednie wyniki są tylko etapami na drodze do ostatecznego rezultatu.

Łańcuchy metod można stosować wszędzie tam, gdzie interfejs API generuje obiekty z tym samym interfejsem (zobacz sposób 57.) za pomocą metod zwracających obiekty (często są one częścią tego interfejsu). Metody służące do przechodzenia po elementach tablicy opisane w sposobach 50. i 51. to następny świetny przykład interfejsu API umożliwiającego zastosowanie łańcucha wywołań.

```
var users = records.map(function(record) {
  return record.username;
})
.filter(function(username) {
  return !!username;
})
.map(function(username) {
  return username.toLowerCase();
});
```

Te połączone w łańcuch operacje przyjmują tablicę obiektów reprezentujących rekordy użytkownika, pobierają z każdego rekordu właściwość z nazwą użytkownika, odfiltrowują puste właściwości i przekształcają znaki w nazwach na małe.

Ten styl zapewnia użytkownikom danego interfejsu API tak dużą swobodę i jest tak zwarty, że warto projektować własne interfejsy API z myślą o nim. W bezstanowych interfejsach API często możliwość łączenia w łańcuch jest naturalna. Jeśli dany interfejs API nie modyfikuje obiektu, musi zwrócić nowy obiekt. W efekcie powstaje interfejs API, którego metody generują obiekty z podobnym zestawem metod.

Łańcuchy metod są też przydatne w rozwiązaniach z obsługą stanu. Sztuczka polega na tym, aby metody aktualizujące obiekt zwracały obiekt `this` zamiast wartości `undefined`. Pozwala to wprowadzić kilka zmian w tym samym obiekcie w sekwencji wywołań metod połączonych w łańcuch.

```
element.setBackgroundColor("yellow")
    .setColor("red")
    .setFontWeight("bold");
```

Łańcuchy metod w stanowych interfejsach API są czasem nazywane **stylem płynnym** (ang. *fluent style*; to określenie wymyślili programiści symulujący sekwencje metod z języka Smalltalk — jest to wbudowana składnia umożliwiająca wywołanie wielu metod jednego obiektu). Jeśli metoda aktualizująca stan nie zwraca obiektu `this`, użytkownik interfejsu API za każdym razem musi ponownie podać nazwę obiektu. Jeżeli nazwa obiektu jest podawana za pomocą zmiennej, zastosowane rozwiązanie nie ma dużego znaczenia. Jednak gdy metody bezstanowe przyjmujące obiekty są używane razem z metodami zmieniającymi stan, łańcuchy metod pozwalają pisać bardzo zwięzły i czytelny kod. To podejście spopularyzowano w służącej do tworzenia frontonów witryn bibliotece jQuery, gdzie dostępny jest zestaw (bezstanowych) metod do pobierania elementów interfejsu użytkownika ze stron internetowych i zestaw (stanowych) metod do aktualizowania tych elementów.

```
$("#notification") // Wyszukiwanie elementu notification
    .html("Serwer nie odpowiada.") // Ustawianie komunikatu
    .removeClass("info") // Usuwanie jednego stylu
    .addClass("error"); // Dodawanie innego stylu
```

Ponieważ stanowe wywołania metod `html`, `removeClass` i `addClass` są zgodne z płynnym stylem, bo zwracają ten sam obiekt, nie trzeba nawet tworzyć zmiennej tymczasowej na wynik pobrania elementu przez funkcję `$` biblioteki jQuery. Oczywiście jeśli użytkownik stwierdzi, że ten styl jest zbyt lakoniczny, zawsze może dodać zmienną i nazwać w ten sposób wynik pobierania elementu.

```
var element = $("#notification");
element.html("Serwer nie odpowiada.");
element.removeClass("info");
element.addClass("error");
```

Jednak dzięki obsłudze łańcuchów metod interfejs API umożliwia programistom zdecydowanie, który styl preferują. Gdyby metody zwracały wartość `undefined`, użytkownicy zawsze musieliby pisać kod w bardziej rozwlekłej postaci.



### Co warto zapamiętać?

- Stosuj łańcuchy metod do łączenia bezstanowych operacji.
- Zapewnij obsługę łańcuchów metod. W tym celu projektuj bezstanowe metody zwracające nowe obiekty.
- W metodach stanowych zapewnij obsługę łańcuchów metod przez zwracanie obiektu `this`.



# 7

## Współbieżność

JavaScript został zaprojektowany jako **osadzany język skryptowy**. Programy napisane w JavaScriptcie nie są uruchamiane jako niezależne aplikacje, ale jako skrypty w większych aplikacjach. Najbardziej znanym przykładem są oczywiście przeglądarki internetowe. W przeglądarce otwartych może być wiele okien i zakładek, w których działają liczne aplikacje internetowe. Każda z nich reaguje na różne dane wyjściowe i sygnały — działania użytkownika odbierane za pomocą klawiatury, myszy lub touchpada, dane przychodzące z sieci lub alarmy o ustalonych godzinach. Te zdarzenia mogą zachodzić w dowolnym momencie (także jednocześnie) pracy aplikacji. Aplikacja może pobierać informacje o poszczególnych zdarzeniach i reagować na nie odpowiednimi operacjami.

Sposób pisania programów reagujących na wiele równoległych zdarzeń w JavaScriptcie jest niezwykle wygodny dla użytkowników i daje dużo możliwości. Wykorzystywany jest w nim prosty model wykonywania kodu (nazywany **kolejką zdarzeń** lub **współbieżnością opartą na pętli zdarzeń**) razem z *asynchronicznymi* interfejsami API. Ponieważ to podejście działa bardzo efektywnie, a standard języka JavaScript jest rozwijany niezależnie od przeglądarek internetowych, JavaScript można stosować jako język programowania dla rozmaitych aplikacji — od programów desktopowych po platformy działające po stronie serwera (takie jak Node.js).

Co ciekawe, w standardzie ECMAScript do tej pory nie ma nic na temat współbieżności. Dlatego w tym rozdziale opisane są rzeczywiste cechy JavaScriptu, a nie oficjalny standard tego języka. Mimo to w większości środowisk JavaScriptu współbieżność jest obsługiwana w ten sam sposób. Przyszłe wersje standardu mogą opisywać ten powszechnie przyjęty model działania. Niezależnie od standardu korzystanie ze zdarzeń i asynchronicznych interfejsów API to ważne aspekty programowania w JavaScriptcie.

## Sposób 61. Nie blokuj kolejki zdarzeń operacjami wejścia-wyjścia

Struktura programów w JavaScriptcie jest oparta na **zdarzeniach**, czyli na danych wejściowych (mogą one nadchodzić jednocześnie) pochodzących z różnych zewnętrznych źródeł. Zdarzenia są generowane przez interakcje z użytkownikiem (po kliknięciu przycisku myszy, wciśnięciu klawisza lub dotknięciu ekranu), przychodzące dane sieciowe lub alarmy uruchamiane o ustalonej godzinie. W niektórych językach typowym rozwiązaniem jest pisanie kodu, który oczekuje na określone dane wejściowe.

```
var text = downloadSync("http://example.com/file.txt");
console.log(text);
```

Interfejs API `console.log` to typowe narzędzie w platformach JavaScriptu służące do wyświetlania informacji diagnostycznych w konsoli programisty. Funkcje takie jak `downloadSync` są nazywane **synchronicznymi** lub **blokującymi**. Program w trakcie oczekiwania na dane wejściowe wstrzymuje pracę. Tu te dane to wynik pobierania pliku przez internet. Ponieważ komputer może wykonywać inne przydatne zadania w trakcie oczekiwania na zakończenie pobierania, takie języki zwykle zapewniają programiście mechanizm tworzenia wielu **wątek**. Odpowiadają one za dodatkowe, jednocześnie wykonywane obliczenia. Dzięki temu jedna część programu może wstrzymać pracę i oczekiwać (zablokowana) na wolno napływające dane wejściowe, podczas gdy druga część kontynuuje wykonywanie przydatnych niezależnych zadań.

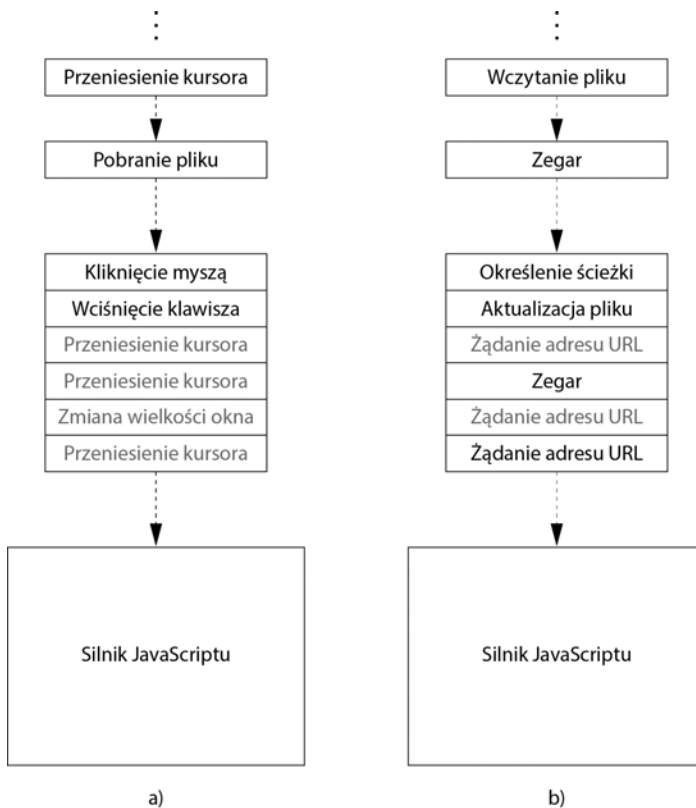
W JavaScriptcie większość operacji wejścia-wyjścia jest obsługiwana za pomocą **asynchronicznych (nieblokujących)** interfejsów API. Zamiast blokować wątek w oczekiwaniu na wynik, programista udostępnia wywołanie zwrotne (zobacz sposób 19.), wywoływane przez system w momencie odebrania danych wejściowych.

```
downloadAsync("http://example.com/file.txt", function(text) {
    console.log(text);
});
```

Zamiast blokować program w oczekiwaniu na dane z sieci, ten interfejs API inicjuje proces pobierania, a następnie natychmiast zwraca sterowanie po zapisaniu wywołania zwrotnego w wewnętrznym rejestrze. Później, gdy pobieranie zostanie ukończone, system uruchamia zarejestrowane wywołanie zwrotne i przekazuje do niego tekst z pobranego pliku jako argument.

System nie uruchamia wywołania zwrotnego natychmiast po zakończeniu pobierania. JavaScript zapewnia gwarancję *wykonywania aż do ukończenia*. Dlatego kod użytkownika wykonywany obecnie we współużytkowanym kontekście (na przykład na jednej stronie internetowej w przeglądarce lub na jednym egzemplarzu serwera WWW) może zakończyć pracę przed uruchomieniem następnej metody obsługi zdarzeń. System przechowuje wewnętrzną kolejkę zdarzeń i uruchamia zarejestrowane wywołania zwrotne jedno po drugim.

Rysunek 7.1 przedstawia przykładową kolejkę zdarzeń w aplikacjach działających po stronie klienta i po stronie serwera. Gdy nastąpi zdarzenie, jest ono dodawane na koniec kolejki zdarzeń aplikacji (górna część ekranu). System JavaScriptu wykonuje aplikację w wewnętrznej **pętli zdarzeń**, która pobiera zdarzenia z dolnej części kolejki (czyli w kolejności ich następowania) i uruchamia jedną po drugiej zarejestrowane metody obsługi zdarzeń w języku JavaScript. Są to wywołania zwrotne, takie jak wywołanie przekazane w przykładowym kodzie do funkcji `downloadAsync`. Do metod obsługi zdarzeń jako argumenty przekazywane są dane dotyczące zdarzenia.



**Rysunek 7.1.** Przykładowe kolejki zdarzeń w internetowej aplikacji klienckiej (a) i na serwerze WWW (b)

Zaletą gwarancji wykonywania aż do ukończenia zadania jest to, że gdy kod działa, wiadomo, że ma kompletną kontrolę nad stanem aplikacji. Programista nie musi się martwić, że wartość zmiennej lub właściwość obiektu zmieni się z powodu równoległe wykonywanego kodu. Korzystnym efektem tego stanu rzeczy jest to, że programowanie współbieżne w JavaScriptcie jest znacznie łatwiejsze niż korzystanie z wątków i blokad w takich językach jak C++, Java lub C#.

Wadą wykonywania aż do ukończenia zadania jest to, że działający kod wstrzymuje pracę reszty aplikacji. W aplikacjach interaktywnych (na przykład w przeglądarce) blokująca metoda obsługi zdarzeń uniemożliwia obsługę innych danych wejściowych od użytkownika i może nawet zablokować wyświetlanie strony. Użytkownik ma wtedy wrażenie, że strona nie reaguje. Na serwerze blokująca metoda obsługi zdarzeń może uniemożliwić obsługę innych żądań sieciowych. Wtedy to serwer wygląda tak, jakby nie reagował na żądania.

Najważniejszą regułą programowania współbieżnego w JavaScriptcie jest to, aby nigdy nie używać blokujących interfejsów API dla operacji wejścia-wyjścia dodawanych do kolejki zdarzeń aplikacji. W przeglądarce liczba blokujących interfejsów API jest bardzo mała, choć przez lata kilka takich elementów niestety się pojawiło. Biblioteka `XMLHttpRequest`, obsługująca sieciowe operacje wejścia-wyjścia podobne do funkcji `downloadAsync`, posiada synchroniczną wersję uznawaną za niezalecane rozwiązanie. Synchroniczne operacje wejścia-wyjścia mają poważny negatywny wpływ na interaktywność aplikacji sieciowych, ponieważ uniemożliwiają użytkownikom interakcję ze stroną do czasu zakończenia operacji wejścia-wyjścia.

Natomiast asynchroniczne interfejsy API są bezpieczne w użyciu razem ze zdarzeniami, ponieważ powodują, że kod aplikacji kontynuuje działanie w kolejnych powtórzeniach pętli zdarzeń. Załóżmy, że pobranie strony o danym adresie URL zajmuje kilka sekund. W tym czasie może nastąpić wiele innych zdarzeń. W środowisku synchronicznym te zdarzenia zbierają się w kolejce zdarzeń, przy czym jest ona zablokowana do czasu zakończenia pracy przez kod w JavaScriptcie. Przetwarzanie innych zdarzeń jest więc niemożliwe. Jednak w wersji asynchronicznej kod w JavaScriptcie rejestruje metodę obsługi zdarzeń i natychmiast zwraca sterowanie. Inne metody obsługi zdarzeń mogą następnie przetworzyć kolejne zdarzenia przed zakończeniem pobierania strony.

Gdy blokujące operacje nie wpływają na główną kolejkę zdarzeń aplikacji, stanowią mniejszy problem. Na przykład w klientach sieciowych dostępny jest interfejs API w postaci obiektów `Worker`, które umożliwiają uruchamianie współbieżnych obliczeń. Obiekty `Worker`, w odróżnieniu od standardowych wątków, są wykonywane w izolacji, bez dostępu do globalnego zasięgu lub zawartości stron z głównego wątku aplikacji. Dlatego nie zakłócają pracy kodu związanego z główną kolejką zdarzeń. W obiektach `Worker` stosowanie synchronicznej wersji obiektu `XMLHttpRequest` sprawia mniej kłopotów. Zablokowanie pracy na czas pobierania danych uniemożliwia dalszą pracę danego obiektu `Worker`, ale nie przeszkadza w wyświetlaniu strony lub reagowaniu na zdarzenia z kolejki zdarzeń. Na serwerze blokujące interfejsy API nie są problemem w trakcie rozruchu, czyli przed rozpoczęciem reagowania przez serwer na przychodzące żądania. Jednak na etapie obsługi żądań blokujące interfejsy API są równie problematyczne jak w kolejce zdarzeń w przeglądarce.

## Co warto zapamiętać?

- Asynchroniczne interfejsy API przyjmują wywołania zwrotne, co pozwala odsunąć w czasie przetwarzanie czasochłonnych operacji i uniknąć zablokowania głównej aplikacji.
- JavaScript przyjmuje zdarzenia równolegle, ale wykonuje metody obsługi zdarzeń sekwencyjnie, z wykorzystaniem kolejki zdarzeń.
- Nigdy nie stosuj blokujących operacji wejścia-wyjścia w kolejce zdarzeń aplikacji.

## Sposób 62. Stosuj zagnieżdżone lub nazwane wywołania zwrotne do tworzenia sekwencji asynchronicznych wywołań

W sposobie 61. pokazano, że asynchroniczne interfejsy API pozwalają wykonywać potencjalnie kosztowne operacje wejścia-wyjścia bez blokowania pracy aplikacji i przetwarzania innych danych wejściowych. Zrozumienie kolejności operacji w programach asynchronicznych bywa początkowo skomplikowane. Na przykład poniższy program wyświetla tekst "rozpoczęcie" przed słowem "zakończenie", choć w kodzie źródłowym instrukcje wyświetlające te słowa występują w odwrotnej kolejności.

```
downloadAsync("file.txt", function(file) {  
    console.log("zakończenie");  
});  
console.log("rozpoczęcie");
```

Wywołanie `downloadAsync` natychmiast zwraca sterowanie — bez oczekiwania na zakończenie pobierania pliku. Ponieważ w JavaScriptcie gwarantowane jest wykonywanie zadań aż do ukończenia, następny wiersz kodu zostaje wykonany przed uruchomieniem innych metod obsługi zdarzeń. Dlatego wiadomo, że słowo "rozpoczęcie" pojawi się przed słowem "zakończenie".

Aby zrozumieć tę sekwencję operacji, najlepiej wyobrazić sobie, że asynchroniczny interfejs API *inicjuje* operację, zamiast ją *wykonywać*. Przedstawiony kod najpierw inicjuje pobieranie pliku, a następnie natychmiast wyświetla słowo "rozpoczęcie". Gdy pobieranie zostanie ukończony, w odrębnym powtórzeniu pętli zdarzeń zarejestrowana metoda obsługi zdarzeń wyświetli słowo "zakończenie".

Tak więc skoro umieszczenie kilku instrukcji po kolei sprawdza się tylko wtedy, gdy trzeba wykonać jakiś zadanie po zainicjowaniu operacji, jak uporządkować w pełni asynchroniczne operacje? Co zrobić, jeśli trzeba sprawdzić adres URL w asynchronicznie działającej bazie danych, a następnie pobrać

zawartość strony o tym adresie? Nie da się zainicjować obu tych żądań jedno po drugim.

```
db.lookupAsync("url", function(url) {
    // ?
});
downloadAsync(url, function(text) { // Błąd: zmienna url nie jest związana z wartością
    console.log("Zawartość strony " + url + ": " + text);
});
```

To nie zadziała, ponieważ w funkcji `downloadAsync` potrzebny jest argument w postaci adresu URL pobranego z bazy, który jest jednak niedostępny. Jest to zrozumiałe. Na razie program tylko zainicjował wyszukiwanie adresu w bazie danych. Wynik tej operacji nie jest jeszcze dostępny.

Najprostszym rozwiązaniem jest zagnieżdżanie. Dzięki domknięciom (zobacz sposób 11.) można zagnieżdżyć drugą operację w wywołaniu zwrotnym pierwszej.

```
db.lookupAsync("url", function(url) {
    downloadAsync(url, function(text) {
        console.log("Zawartość strony " + url + ": " + text);
    });
});
```

Nadal występują tu dwa wywołania zwrotne, przy czym drugie jest umieszczone w pierwszym. Powstaje w ten sposób domknięcie mające dostęp do zmiennych z zewnętrznego wywołania zwrotnego. Zauważ, że w drugim wywołaniu zwrotnym używana jest zmienna `url`.

Zagnieżdżanie asynchronicznych operacji jest proste, jednak gdy sekwencje są dłuższe, kod staje się mało czytelny.

```
db.lookupAsync("url", function(url) {
    downloadAsync(url, function(file) {
        downloadAsync("a.txt", function(a) {
            downloadAsync("b.txt", function(b) {
                downloadAsync("c.txt", function(c) {
                    // ...
                });
            });
        });
    });
});
```

Jednym ze sposobów na ograniczenie nadmiernego zagnieżdżania jest przeniesienie zagnieżdżonych wywołań zwrotnych do funkcji nazwanych i przekazywanie do nich dodatkowych danych za pomocą argumentów. Wcześniejszy dwuetapowy przykład można w tym podejściu zapisać w następujący sposób:

```
db.lookupAsync("url", downloadURL);

function downloadURL(url) {
    downloadAsync(url, function(text) { // Wywołanie nadal jest zagnieżdżone
```



```

        showContents(url, text);
    });
}
function showContents(url, text) {
    console.log("Zawartość strony " + url + ": " + text);
}

```

W funkcji `downloadURL` nadal znajduje się zagnieżdżone wywołanie zwrotne, co pozwala połączyć zewnętrzną zmienną `url` z wewnętrzną zmienną `text` i przekazać je razem jako argumenty funkcji `showContents`. Ostatnie zagnieżdżone wywołanie zwrotne można wyeliminować za pomocą instrukcji `bind` (zobacz sposób 25.).

```

db.lookupAsync("url", downloadURL);

function downloadURL(url) {
    downloadAsync(url, showContents.bind(null, url));
}

function showContents(url, text) {
    console.log("Zawartość strony " + url + ": " + text);
}

```

W tym modelu kod wygląda na bardziej sekwencyjny, jednak dzieje się to kosztem konieczności nazwania każdego pośredniego kroku w sekwencji i kopiowania odpowiednich zmiennych między krokami. Bywa to kłopotliwe w przypadku długich sekwencji (takich jak przedstawiona wcześniej).

```

db.lookupAsync("url", downloadURLAndFiles);

function downloadURLAndFiles(url) {
    downloadAsync(url, downloadABC.bind(null, url));
}

// Dziwna nazwa
function downloadABC(url, file) {
    downloadAsync("a.txt",
        // Zduplikowane wiązania
        downloadFiles23.bind(null, url, file));
}

// Dziwna nazwa
function downloadBC(url, file, a) {
    downloadAsync("b.txt",
        // Więcej zduplikowanych wiązań
        downloadFile3.bind(null, url, file, a));
}

// Dziwna nazwa
function downloadC(url, file, a, b) {
    downloadAsync("c.txt",
        // Jeszcze więcej zduplikowanych wiązań
        finish.bind(null, url, file, a, b));
}

```

```
function finish(url, file, a, b, c) {
    // ...
}
```

Czasem połączenie obu tych podejść daje lepsze efekty, choć konieczne jest zagnieżdżanie.

```
db.lookupAsync("url", function(url) {
    downloadURLAndFiles(url);
});

function downloadURLAndFiles(url) {
    downloadAsync(url, downloadFiles.bind(null, url));
}

function downloadFiles(url, file) {
    downloadAsync("a.txt", function(a) {
        downloadAsync("b.txt", function(b) {
            downloadAsync("c.txt", function(c) {
                // ...
            });
        });
    });
}
```

Ostatni krok można jeszcze usprawnić, dodając mechanizm pobierania wielu plików i zapisywania ich w tablicy.

```
function downloadFiles(url, file) {
    downloadAllAsync(["a.txt", "b.txt", "c.txt"],
        function(all) {
            var a = all[0], b = all[1], c = all[2];
            // ...
        });
}
```

Użycie funkcji `downloadAllAsync` umożliwia równoległe pobieranie wielu plików. W modelu sekwencyjnym operacji nie można nawet zainicjować przed zakończeniem poprzedniej. Niektóre operacje są z natury sekwencyjne — na przykład pobieranie strony o adresie URL wczytanym z bazy danych. Jeśli jednak dostępna jest lista nazw plików do pobrania, zapewne nie trzeba czekać na zakończenie pobierania każdego pliku przed rozpoczęciem wczytywania następnych. W sposobie 66. wyjaśniono, jak implementować współbieżne rozwiązania takie jak funkcja `downloadAllAsync`.

Oprócz zagnieżdżania wywołań zwrotnych i nadawania im nazw można też budować mechanizmy wyższego poziomu, aby uprościć przepływ sterowania asynchronicznymi wywołaniami i móc robić to w bardziej zwięzły sposób. Bardzo popularne podejście tego rodzaju opisano w sposobie 68. Warto też zapoznać się z asynchronicznymi bibliotekami i przeprowadzić własne próby z różnymi mechanizmami.

### Co warto zapamiętać?

- Stosuj zagnieżdżone lub nazwane wywołania zwrotne, aby wykonywać kilka asynchronicznych operacji po kolei.
- Staraj się zachować równowagę między nadmiernym zagnieżdżaniem wywołań zwrotnych a dziwnymi nazwami niezagnieżdżonych wywołań zwrotnych.
- Unikaj sekwencyjnego wykonywania operacji, które można przeprowadzać współbieżnie.

### Sposób 63. Pamiętaj o ignorowanych błędach

Jednym ze skomplikowanych aspektów programowania asynchronicznego jest zarządzanie obsługą błędów. W kodzie synchronicznym łatwo jest obsługiwać błędy w jednym miejscu. Wystarczy w tym celu umieścić sekcję kodu w bloku `try`.

```
try {  
    f();  
    g();  
    h();  
} catch (e) {  
    // Obsługa wszystkich zgłoszonych błędów  
}
```

W kodzie asynchronicznym wieloetapowy proces zwykle jest podzielony na odrębne elementy z kolejki zdarzeń, dlatego nie da się ich wszystkich umieścić w jednym bloku `try`. Co więcej, asynchroniczne interfejsy API w ogóle nie mogą zgłaszać błędów, ponieważ w momencie ich wystąpienia nie istnieje kontekst wykonania, do którego można przekazać dany błąd. W asynchronicznych interfejsach API błędy są zwykle reprezentowane jako specjalne argumenty wywołań zwrotnych. Można też stosować dodatkowe wywołania zwrotne związane z obsługą błędów (ang. *errback*). Na przykład asynchroniczny interfejs API służący do pobierania pliku (taki jak pokazany w sposobie 61.) może przyjmować dodatkową funkcję wywoływaną w reakcji na błąd sieci.

```
downloadAsync("http://example.com/file.txt", function(text) {  
    console.log("Zawartość pliku: "+ text);  
}, function(error) {  
    console.log("Błąd: "+ error);  
});
```

W celu pobrania kilku plików można zagnieżdżyć wywołania zwrotne w sposób wyjaśniony w sposobie 62.

```
downloadAsync("a.txt", function(a) {  
    downloadAsync("b.txt", function(b) {  
        downloadAsync("c.txt", function(c) {
```

```

        console.log("Zawartość: " + a + b + c);
    }, function(error) {
        console.log("Błąd: " + error);
    });
}, function(error) { // Powtarzający się kod do obsługi błędów
    console.log("Błąd: " + error);
});
}, function(error) { // Powtarzający się kod do obsługi błędów
    console.log("Error: " + error);
});
};

```

Zauważ, że w każdym kroku procesu używany jest ten sam kod do obsługi błędów, przy czym trzeba umieścić go w kilku miejscach. Jak zawsze w programowaniu warto dążyć do eliminowania powtórzeń. Łatwo można przenieść potrzebny kod do funkcji obsługi błędów dostępnej we wspólnym zasięgu.

```

function onError(error) {
    console.log("Błąd: " + error);
}

downloadAsync("a.txt", function(a) {
    downloadAsync("b.txt", function(b) {
        downloadAsync("c.txt", function(c) {
            console.log("Zawartość: " + a + b + c);
        }, onError);
    }, onError);
}, onError);

```

Oczywiście po połączeniu wielu kroków w jedną złożoną operację za pomocą narzędzi takich jak funkcja `downloadAllAsync` (zgodnie z zaleceniami z zagadnień 62. i 66.) wystarczy ustawić tylko jedno wywołanie zwrotne do obsługi błędów.

```

downloadAllAsync(["a.txt", "b.txt", "c.txt"], function(abc) {
    console.log("Zawartość: " + abc[0] + abc[1] + abc[2]);
}, function(error) {
    console.log("Błąd: " + error);
});

```

Inny rodzaj interfejsów API do obsługi błędów, spopularyzowany w platformie Node.js, polega na przyjmowaniu jednego wywołania zwrotnego, w którym pierwszy argument to albo błąd (jeśli wystąpił), albo wartość reprezentująca fałsz (na przykład `null`). W interfejsach API tego typu też można zdefiniować współużytkowaną funkcję do obsługi błędów, przy czym każde wywołanie zwrotne trzeba zabezpieczyć instrukcją `if`.

```

function onError(error) {
    console.log("Błąd: " + error);
}

downloadAsync("a.txt", function(error, a) {
    if (error) {
        onError(error);
        return;
    }
}

```

```

downloadAsync("b.txt", function(error, b) {
    // Powtarzający się kod do obsługi błędów
    if (error) {
        onError(error);
        return;
    }
    downloadAsync(url3, function(error, c) {
        // Powtarzający się kod do obsługi błędów
        if (error) {
            onError(error);
            return;
        }
        console.log("Zawartość: " + a + b + c);
    });
});
});

```

W platformach z tego rodzaju wywołaniami zwrotnymi do obsługi błędów programiści często rezygnują z konwencji, zgodnie z którą instrukcje `if` powinny zajmować kilka wierszy kodu umieszczonych w nawiasie klamrowym. W efekcie powstaje bardziej zwięzły i mniej rozpraszający kod do obsługi błędów.

```

function onError(error) {
    console.log("Błąd: " + error);
}

downloadAsync("a.txt", function(error, a) {
    if (error) return onError(error);

    downloadAsync("b.txt", function(error, b) {
        if (error) return onError(error);

        downloadAsync(url3, function(error, c) {
            if (error) return onError(error);

            console.log("Zawartość: " + a + b + c);
        });
    });
});
});

```

Jak zwykle połączenie kroków z abstrakcją pomaga wyeliminować powtórzenia.

```

var filenames = ["a.txt", "b.txt", "c.txt"];

downloadAllAsync(filenames, function(error, abc) {
    if (error) {
        console.log("Błąd: " + error);
        return;
    }
    console.log("Zawartość: " + abc[0] + abc[1] + abc[2]);
});

```

Istotną z praktycznego punktu widzenia różnicą między blokiem `try...catch` a typową obsługą błędów w asynchronicznych interfejsach API jest to, że blok `try` ułatwia zdefiniowanie kodu przechwytyującego wszystkie błędy. Dlatego trudno jest zapomnieć o obsłudze błędów dla całego obszaru kodu. W asyn-

chronicznych interfejsach API (takich jak pokazany wcześniej) bardzo łatwo jest pominąć obsługę błędów w którymś z kroków. Często prowadzi to do niewykrytego ignorowania błędów. Programy, które ignorują błędy, bywają bardzo frustrujące dla użytkowników. Aplikacja nie udostępnia wtedy informacji zwrotnych o problemach. Czasem prowadzi to do trwałego zatrzymania się paska postępu. Niezauważalne błędy są też bardzo trudne w debugowaniu, ponieważ programista nie ma wtedy żadnych wskazówek co do źródła problemu. Najlepszym rozwiązaniem jest prewencja. Korzystanie z asynchronicznych interfejsów API wymaga staranności i upewnienia się, że wszystkie warunki wystąpienia błędów zostały jawnie obsłużone.

### Co warto zapamiętać?

- Unikaj kopiowania i wklejania kodu do obsługi błędów. Zamiast tego twórz współużytkowane funkcje do obsługi błędów.
- Koniecznie dodaj obsługę wszystkich warunków wystąpienia błędów, aby uniknąć ignorowania problemów przez kod.

### Sposób 64. Stosuj rekurencję do tworzenia asynchronicznych pętli

Wyobraź sobie funkcję, która przyjmuje tablicę adresów URL i próbuje po kolei pobrać odpowiadające im strony. Funkcja ma kończyć pracę po udanym pobraniu pierwszej strony. W synchronicznym interfejsie API łatwo jest zaimplementować taką funkcję za pomocą pętli.

```
function downloadOneSync(urls) {
  for (var i = 0, n = urls.length; i < n; i++) {
    try {
      return downloadSync(urls[i]);
    } catch (e) { }
  }
  throw new Error("Nie udało się pobrać żadnej strony");
}
```

Jednak to podejście nie zadziała dla funkcji `downloadOneAsync`, ponieważ nie można wstrzymać pętli i wznowić jej w wywołaniu zwrotnym. Gdy zastosujesz pętlę, zainicjuje ona wszystkie operacje pobierania, zamiast czekać na pobranie jednej strony przed przejściem do następnej.

```
function downloadOneAsync(urls, onSuccess, onError) {
  for (var i = 0, n = urls.length; i < n; i++) {
    downloadAsync(urls[i], onSuccess, function(error) {
      // ?
    });
  }
}
```

```

        // Pętla kontynuuje działanie
    }
    throw new Error("Nie udało się pobrać żadnej strony");
}

```

Trzeba więc napisać kod, który działa jak pętla, ale nie kontynuuje pracy bez jawnego żądania. Rozwiązanie polega na zaimplementowaniu pętli jako funkcji, dzięki czemu można zdecydować o momencie uruchomienia każdej iteracji.

```

function downloadOneAsync(urls, onSuccess, onFailure) {
    var n = urls.length;

    function tryNextURL(i) {
        if (i >= n) {
            onFailure("Nie udało się pobrać żadnej strony");
            return;
        }
        downloadAsync(urls[i], onSuccess, function() {
            tryNextURL(i + 1);
        });
    }

    tryNextURL(0);
}

```

Lokalna funkcja `tryNextURL` działa **rekurencyjnie**. Jej implementacja obejmuje wywołanie jej samej. W typowych środowiskach JavaScriptu rekurencyjna funkcja, która zbyt wiele razy wywoła samą siebie synchronicznie, może spowodować błąd. Na przykład poniższa prosta funkcja rekurencyjna próbuje wywołać samą siebie 100 000 razy. W większości środowisk JavaScriptu spowoduje to błąd czasu wykonania.

```

function countdown(n) {
    if (n === 0) {
        return "done";
    } else {
        return countdown(n - 1);
    }
}

```

```
countdown(100000); // Błąd – przekroczenie maksymalnej wielkości stosu wywołań
```

Jak zapewnić bezpieczeństwo rekurencyjnej funkcji `downloadOneAsync`, skoro funkcja `countdown` powoduje błąd dla zbyt dużej wartości `n`? Aby rozwiązać ten problem, sprawdźmy komunikat o błędzie zwracany przez funkcję `countdown`.

Środowiska JavaScriptu zwykle rezerwują określony obszar pamięci (nazywany **stosem wywołań**) na śledzenie, co należy zrobić po zwróceniu sterowania przez wywołania funkcji. Wyobraź sobie, że wykonywany jest poniższy prosty program.

```

function negative(x) {
    return abs(x) * -1;
}

```

```
function abs(x) {
    return Math.abs(x);
}

console.log(negative(42));
```

W aplikacji w miejscu wywołania metody `Math.abs` z argumentem 42 w toku jest kilka innych wywołań funkcji oczekujących na zwrócenie sterowania przez jedną z pozostałych. Rysunek 7.2 przedstawia stos wywołań na tym etapie. Kropki (•) ilustrują, gdzie każda funkcja została wywołana i gdzie dane wywołanie po zakończeniu pracy zwróci sterowanie. Podobnie jak w standardowym stosie informacje przepływają zgodnie z kolejką LIFO (ang. *last-in, first-out*, czyli ostatni na wejściu, pierwszy na wyjściu). Tak więc ostatnie wywołanie funkcji zapisane na stosie (reprezentowane przez ostatnią ramkę na rysunku) zostanie jako pierwsze z niego ściągnięte. Gdy funkcja `Math.abs` zakończy pracę, zwróci sterowanie do funkcji `abs`, która przekaże sterowanie do funkcji `negative`, a ta zwróci sterowanie do zewnętrznego skryptu.

(Początek skryptu)	<code>console.log(•);</code>
<code>negative(42)</code>	<code>return times(•, -1);</code>
<code>abs(42)</code>	<code>return •;</code>
<code>Math.abs(42)</code>	[Wbudowany kod]

**Rysunek 7.2.** Stos wywołań w trakcie wykonywania prostego programu

Gdy w programie w danym momencie wykonywanych jest zbyt wiele funkcji, może zabraknąć miejsca na stosie, co prowadzi do zgłoszenia wyjątku. Taka sytuacja to **przepełnienie stosu**. W omawianym przykładzie wywołanie `countdown(100000)` wymaga, aby funkcja `countdown` wywołała samą siebie 100 000 razy. Za każdym razem powoduje to dodanie następnej ramki stosu, co przedstawia rysunek 7.3. Wymaga to tak dużo miejsca, że w większości środowisk JavaScriptu prowadzi do przepełnienia zaalokowanego obszaru i błędu czasu wykonania.

Przyjrzyj się jeszcze raz funkcji `downloadOneAsync`. W odróżnieniu od funkcji `countdown`, która nie może zwrócić sterowania do momentu zakończenia wywołania rekurencyjnego, funkcja `downloadOneAsync` wywołuje samą siebie tylko w asynchronicznym wywołaniu zwrotnym. Pamiętaj, że asynchroniczne interfejsy API natychmiast zwracają sterowanie — jeszcze przed uruchomieniem wywołania zwrotnego. Tak więc funkcja `downloadOneAsync` zwraca sterowanie (co powoduje zdjęcie jej ramki ze stosu wywołań) jeszcze *przed* dodaniem na stos



(Początek skryptu)	<code>console.log(•);</code>
<code>countdown(100000)</code>	<code>return countdown(•);</code>
<code>countdown(99999)</code>	<code>return countdown(•);</code>
<code>countdown(99998)</code>	<code>return countdown(•);</code>
⋮	
<code>countdown(1)</code>	<code>return countdown(•);</code>
<code>countdown(0)</code>	<code>return "done";</code>

**Rysunek 7.3.** Stos wywołań w trakcie wykonywania funkcji rekurencyjnej

nowej ramki przez wywołanie rekurencyjne. Wywołanie zwrotne zawsze jest uruchamiane w odrębnym powtórzeniu pętli zdarzeń. W każdym powtórzeniu pętli zdarzeń przed wywołaniem metod obsługi zdarzeń stos wywołań jest początkowo pusty. Dlatego funkcja `downloadOneAsync` niezależnie od liczby potrzebnych powtórzeń nigdy nie zajmuje zbyt dużo miejsca na stosie wywołań.

### Co warto zapamiętać?

- Pętle nie mogą być wykonywane asynchronicznie.
- Stosuj funkcje rekurencyjne do wykonywania iteracji w odrębnych powtórzeniach pętli zdarzeń.
- Rekurencja wykonywana w odrębnych powtórzeniach pętli zdarzeń nie prowadzi do przepełnienia stosu wywołań.

## Sposób 65. Nie blokuj kolejki zdarzeń obliczeniami

W sposobie 61. wyjaśniono, że asynchroniczne interfejsy API pomagają zapobiec blokowaniu kolejki zdarzeń aplikacji. To jednak nie wszystko. Każdy programista przyzna, że łatwo jest zablokować aplikację nawet jednym wywołaniem funkcji.

```
while (true) { }
```

Ponadto nie trzeba pisać pętli nieskończonych, aby utworzyć wolno działający program. Wykonywanie kodu wymaga czasu, a niewydajne algorytmy i struktury danych mogą prowadzić do długotrwałych obliczeń.

Oczywiście problemy z wydajnością dotyczą nie tylko JavaScriptu. Jednak w programowaniu opartym na zdarzeniach trzeba uwzględnić specjalne ograniczenia. Aby zapewnić wysoką interaktywność aplikacji klienckiej lub zagwarantować, że wszystkie przychodzące żądania zostaną odpowiednio obsłużone w aplikacji serwerowej, trzeba zadbać o to, aby każde powtórzenie pętli zdarzeń było tak krótkie, jak to możliwe. W przeciwnym razie kolejka zdarzeń zacznie się wydłużać, ponieważ metody obsługi zdarzeń nie będą wystarczająco szybko przetwarzać elementów. W przeglądarce kosztowne obliczenia zmniejszają komfort pracy użytkowników, ponieważ w trakcie działania kodu w JavaScriptcie interfejs stron zwykle nie reaguje.

Co więc można zrobić, jeśli aplikacja ma wykonywać kosztowne obliczenia? Nie ma jednej dobrej odpowiedzi na to pytanie, istnieje jednak kilka popularnych technik. Prawdopodobnie najprostszym rozwiązaniem jest zastosowanie mechanizmów obsługi współbieżności, takich jak interfejs API `Worker` z klientów sieciowych. Jest to dobre podejście w grach ze sztuczną inteligencją, w których potrzebne jest sprawdzanie dużej liczby możliwych ruchów. Gra może na samym początku tworzyć wątek roboczy przeznaczony do obliczania ruchów.

```
var ai = new Worker("ai.js");
```

W efekcie powstaje nowy współbieżny wątek wykonawczy z odrębną kolejką zdarzeń. Jego skrypt roboczy to kod źródłowy z pliku *ai.js*. Taki wątek roboczy działa w izolacji. Nie ma bezpośredniego dostępu do żadnych obiektów aplikacji. Jednak aplikacja i wątek roboczy mogą komunikować się, przysyłając między sobą *komunikaty* w postaci łańcuchów znaków. Dlatego gdy gra wymaga, aby komputer wykonał ruch, może przesłać do wątku roboczego komunikat.

```
var userMove = /* ... */;
```

```
ai.postMessage(JSON.stringify({  
  userMove: userMove  
}));
```

Komunikat przekazany jako argument metody `postMessage` jest dodawany do kolejki zdarzeń wątku roboczego. Na potrzeby przetwarzania odpowiedzi od wątku roboczego gra rejestruje metodę obsługi zdarzeń.

```
ai.onmessage = function(event) {  
  executeMove(JSON.parse(event.data).computerMove);  
};
```

Kod z pliku *ai.js* nakazuje wątkowi roboczemu oczekiwać na komunikaty i wykonywać zadania potrzebne do ustalenia następnych ruchów.

```
self.onmessage = function(event) {  
  // Przetwarzanie ruchu użytkownika  
  var userMove = JSON.parse(event.data).userMove;
```

```
// Generowanie następnego ruchu komputera
var computerMove = computeNextMove(userMove);

// Formatowanie ruchu komputera
var message = JSON.stringify({
  computerMove: computerMove
});

self.postMessage(message);
};

function computeNextMove(userMove) {
  // ...
}
```

Nie wszystkie platformy JavaScriptu udostępniają interfejs API taki jak obiekty typu `Worker`. Ponadto czasem koszt przekazywania komunikatów jest zbyt wysoki. Innym rozwiązaniem jest podział algorytmu na kilka kroków; w każdym z nich wykonywana jest akceptowalna ilość pracy. Wróć do algorytmu z listą roboczą ze sposobu 48., który przeszukuje graf w sieci społecznościowej.

```
Member.prototype.inNetwork = function(other) {
  var visited = {};
  var worklist = [this];
  while (worklist.length > 0) {
    var member = worklist.pop();
    // ...
    if (member === other) { // Znaleziono?
      return true;
    }
    // ...
  }
  return false;
};
```

Jeśli używana w tym algorytmie pętla `while` jest zbyt kosztowna, proces przeszukiwania może zablokować kolejkę zdarzeń aplikacji na nieakceptowalnie długi czas. Nawet jeśli dostępne są obiekty typu `Worker`, ich wykorzystanie może okazać się zbyt kosztowne lub niewygodne, ponieważ trzeba albo kopiować cały stan grafu, albo zapisywać stan w wątku roboczym, co wymaga przekazywania komunikatów w celu aktualizowania sieci i pobierania z niej danych.

Przedstawiony algorytm jest na szczęście zdefiniowany jako sekwencja kroków — iteracji pętli `while`. Można przekształcić metodę `inNetwork` na wersję asynchroniczną. W tym celu należy dodać parametr reprezentujący wywołanie zwrotne i, co opisano w sposobie 64., zastąpić pętlę `while` asynchroniczną funkcją rekurencyjną.

```
Member.prototype.inNetwork = function(other, callback) {
  var visited = {};
  var worklist = [this];
  function next() {
```

```

    if (worklist.length === 0) {
        callback(false);
        return;
    }
    var member = worklist.pop();
    // ...
    if (member === other) { // Znaleziono?
        callback(true);
        return;
    }
    // ...
    setTimeout(next, 0); // Planowanie wykonania następnej iteracji
}
setTimeout(next, 0); // Planowanie wykonania następnej iteracji
};

```

Warto szczegółowo opisać działanie tego kodu. Zamiast pętli `while` znajduje się tu lokalna funkcja `next`, która odpowiada za wykonanie jednej iteracji pętli i zaplanowanie następnej wykonywanej asynchronicznie iteracji, umieszczonej w kolejce zdarzeń aplikacji. Dzięki temu inne zdarzenia, które w międzyczasie wystąpiły, mogą zostać przetworzone przed przejściem kodu do następnej iteracji. Po zakończeniu wyszukiwania (albo w wyniku znalezienia elementu, albo po sprawdzeniu całej listy roboczej) uruchamiane jest wywołanie zwrotne z wynikiem. Oznacza to zakończenie pętli, ponieważ funkcja `next` zwraca sterowanie bez planowania dalszych iteracji.

Do planowania iteracji używany jest popularny interfejs API w postaci funkcji `setTimeout`. Jest on dostępny w wielu platformach JavaScriptu i pozwala zarejestrować funkcję `next`, tak aby została uruchomiona po upływie minimalnej ilości czasu (po 0 milisekund). Warto zauważyć, że choć funkcja `setTimeout` jest stosunkowo przenośna między platformami, często dostępne są lepsze rozwiązania. Na przykład w przeglądarkach minimalny czas to cztery milisekundy, natomiast inne techniki, wykorzystujące funkcję `postMessage`, potrafią dodać zdarzenie do kolejki natychmiast.

Jeśli wykonywanie tylko jednej iteracji algorytmu przy poszczególnych przebiegach przez kolejkę zdarzeń aplikacji to za mało, można zmodyfikować algorytm, tak by wykonywał za każdym razem określoną liczbę iteracji. Ten efekt można łatwo uzyskać za pomocą prostej pętli z licznikiem umieszczonej wokół głównej części funkcji `next`.

```

Member.prototype.inNetwork = function(other, callback) {
    // ...
    function next() {
        for (var i = 0; i < 10; i++) {
            // ...
        }
        setTimeout(next, 0);
    }
    setTimeout(next, 0);
};

```

## Co warto zapamiętać?

- Unikaj kosztownych algorytmów w głównej kolejce zdarzeń.
- W platformach, które obsługują interfejs API w postaci obiektów `Worker`, można wykorzystać go do uruchamiania długich obliczeń w odrębnej kolejce zdarzeń.
- Gdy obiekt `Worker` nie jest dostępny (lub gdy używanie go jest zbyt kosztowne), pomyśl o rozbiciu obliczeń tak, by były wykonywane w kolejnych przebiegach pętli zdarzeń.

## Sposób 66. Wykorzystaj licznik do wykonywania współbieżnych operacji

W sposobie 63. opisano, że funkcja narzędzia `downloadAllAsync` może pobierać tablicę adresów URL i wczytywać wszystkie odpowiadające im strony, a następnie zwracać tablicę z zawartością plików (po jednym łańcuchu znaków na każdy adres URL). Ważną zaletą funkcji `downloadAllAsync` jest (oprócz likwidacji zagnieżdżonych wywołań zwrotnych) **współbieżne** pobieranie plików. Zamiast czekać na zakończenie pobierania poszczególnych plików, można jednocześnie zainicjować wszystkie operacje w jednym przebiegu pętli zdarzeń.

Kod współbieżny jest skomplikowany i łatwo popełnić w nim błędy. Oto implementacja z trudną do zauważenia drobną usterką.

```
function downloadAllAsync(urls, onsuccess, onerror) {
    var result = [], length = urls.length;

    if (length === 0) {
        setTimeout(onsuccess.bind(null, result), 0);
        return;
    }

    urls.forEach(function(url) {
        downloadAsync(url, function(text) {
            if (result) {
                // Warunek wyścigu
                result.push(text);
                if (result.length === urls.length) {
                    onsuccess(result);
                }
            }
        }, function(error) {
            if (result) {
                result = null;
                onerror(error);
            }
        });
    });
}
```

W tej funkcji kryje się poważny błąd. Warto jednak zacząć od omówienia jej działania. Najpierw należy się upewnić, że jeśli wejściowa tablica jest pusta, wywołanie zwrótnie jest uruchamiane z pustą wynikową tablicą. W przeciwnym razie żadne z dwóch wywołań zwrótnych nigdy nie zostanie uruchomione, ponieważ pętla metody `forEach` będzie pusta. W sposobie 67. wyjaśniono, dlaczego wywołanie zwrótnie `onsuccess` jest uruchamiane za pomocą funkcji `setTimeout`, a nie bezpośrednio. Następnie kod przechodzi po tablicy adresów URL i żądania asynchronicznego pobrania strony odpowiadającej każdemu z nich. Po każdym udanym pobraniu pliku jego zawartość jest dodawana do tablicy `result`. Po udanym pobraniu wszystkich plików uruchamiana jest wywoływana zwrótnie funkcja `onsuccess` z kompletną tablicą `result`. Jeśli pobieranie któregoś z plików zakończy się niepowodzeniem, uruchamiana jest wywoływana zwrótnie funkcja `onerror` z wartością błędu. Ponieważ niepowodzeniem może zakończyć się próba pobrania wielu plików, po pierwszym błędzie tablica `result` jest ustawiana na wartość `null`, dzięki czemu funkcja `onerror` jest wywoływana tylko raz (dla pierwszego błędu).

Aby zrozumieć, gdzie mogą wystąpić problemy, przyjrzyj się poniższemu przykładowi.

```
var filenames = [
    "huge.txt", // Bardzo duży plik
    "tiny.txt", // Mały plik
    "medium.txt" // Plik średniej wielkości
];
downloadAllAsync(filenames, function(files) {
    console.log("Duży plik: " + files[0].length); // Mały
    console.log("Mały plik: " + files[1].length); // Średni
    console.log("Średni plik: " + files[2].length); // Duży
}, function(error) {
    console.log("Błąd: " + error);
});
```

Ponieważ pliki są pobierane współbieżnie, zdarzenia mogą zachodzić (i trafiać do kolejki zdarzeń aplikacji) w dowolnej kolejności. Jeśli program najpierw pobierze plik *tiny.txt*, potem *medium.txt*, a następnie *huge.txt*, wywołania zwrótnie z funkcji `downloadAllAsync` zostaną uruchomione w innej kolejności, niż zostały dodane. Jednak funkcja `downloadAllAsync` umieszcza wyniki pośrednie na końcu tablicy `result` bezpośrednio po ich odebraniu. Dlatego funkcja generuje tablicę zawierającą pobrane pliki w nieznanej kolejności. Z takiego interfejsu API nie da się poprawnie korzystać, ponieważ jednostka wywołująca nie potrafi określić, który wynik jest który. Pokazany przykład, w którym założono, że wyniki są podawane w takiej samej kolejności jak w wejściowej tablicy, zwróci więc nieprawidłowe informacje.

W sposobie 48. opisano rozwiązania niedeterministyczne. Są to operacje o nieokreślonym wyniku, a na ich rezultatach nie można polegać. Zdarzenia współ-

bieżne to najważniejsze źródło niedeterminizmu w JavaScriptcie. *Kolejność występowania zdarzeń* przy poszczególnych uruchomieniach aplikacji nie jest gwarantowana.

Gdy aplikacja do poprawnego działania wymaga określonej kolejności zdarzeń, jest narażona na zjawisko **wyścigu do danych** (ang. *data race*). Wiele współbieżnych operacji może w inny sposób modyfikować współużytkowaną strukturę danych w zależności od kolejności ich wykonywania. Współbieżne operacje „ścigają” się ze sobą na drodze do ukończenia pracy. Wyścig do danych prowadzi do bardzo frustrujących błędów. Mogą być one niewidoczne w konkretnym przebiegu testu, ponieważ dwa uruchomienia programu mogą prowadzić do innych wyników. Użytkownik funkcji `downloadAllAsync` może na przykład zmienić kolejność plików w taki sposób, aby była zgodna z oczekiwanym porządkiem zakończenia ich pobierania.

```
downloadAllAsync(filenames, function(files) {  
    console.log("Mały plik: " + files[1].length);  
    console.log("Średni plik: " + files[2].length);  
    console.log("Duży plik: " + files[0].length);  
}, function(error) {  
    console.log("Błąd: " + error);  
});
```

Teraz w większości sytuacji wyniki będą pojawiać się w tej samej kolejności. Jednak czasem, na przykład z powodu zmiany obciążenia serwera lub zawartości pamięci podręcznej sieci, pliki mogą zostać pobrane w innym porządku. Diagnoza błędów tego rodzaju sprawia dużo problemów, ponieważ niezwykle trudno jest odtworzyć takie usterki. Oczywiście zawsze można wrócić do sekwencyjnego pobierania plików, co jednak oznacza rezygnację z korzyści, jakie zapewnia współbieżność.

Rozwiązanie polega na zaimplementowaniu funkcji `downloadAllAsync` w taki sposób, aby zawsze zwracała przewidywalne wyniki — niezależnie od nieprzewidywalnej kolejności zdarzeń. Zamiast umieszczać każdy wynik na końcu tablicy, można zapisać go zgodnie z pierwotnym indeksem.

```
function downloadAllAsync(urls, onsuccess, onerror) {  
    var length = urls.length;  
    var result = [];  
  
    if (length === 0) {  
        setTimeout(onsuccess.bind(null, result), 0);  
        return;  
    }  
  
    urls.forEach(function(url, i) {  
        downloadAsync(url, function(text) {  
            if (result) {  
                result[i] = text; // Zapisywanie zgodnie ze stałym indeksem  
            }  
        });  
    });  
}
```

```

        // Sytuacja wyścigu
        if (result.length === urls.length) {
            onsuccess(result);
        }
    }, function(error) {
        if (result) {
            result = null;
            onerror(error);
        }
    });
});
}

```

W tej implementacji wykorzystano drugi argument wywołania zwrótnego metody `forEach`. Ten argument to indeks tablicy odpowiadający bieżącej iteracji. Niestety, także ta wersja kodu nie jest poprawna. W sposobie 51. opisano zasady aktualizowania tablicy. Ustawienie indeksowanej właściwości zawsze gwarantuje, że właściwość `length` tablicy będzie miała wartość większą niż dany indeks. Załóżmy, że wywoływana jest poniższa instrukcja:

```
downloadAllAsync(["huge.txt", "medium.txt", "tiny.txt"]);
```

Jeśli plik *tiny.txt* zostanie pobrany przed pozostałymi plikami, w wynikowej tablicy znajdzie się właściwość o indeksie 2. To spowoduje, że właściwość `result.length` przyjmie wartość 3. Wtedy wywołanie zwrótnie odpowiadające powodzeniu operacji zostanie uruchomione przedwcześnie — zanim tablica wyników się zapełni.

W poprawnej implementacji należy zastosować licznik do śledzenia liczby operacji w toku.

```

function downloadAllAsync(urls, onsuccess, onerror) {
    var pending = urls.length;
    var result = [];

    if (pending === 0) {
        setTimeout(onsuccess.bind(null, result), 0);
        return;
    }

    urls.forEach(function(url, i) {
        downloadAsync(url, function(text) {
            if (result) {
                result[i] = text; // Zapisanie stałego indeksu
                pending--;       // Odnótowanie powodzenia
                if (pending === 0) {
                    onsuccess(result);
                }
            }
        }, function(error) {
            if (result) {
                result = null;
                onerror(error);
            }
        });
    });
}

```



```

    }
  });
}

```

Teraz niezależnie od kolejności zdarzeń licznik `pending` poprawnie określa moment zakończenia obsługi wszystkich zdarzeń. Kompletne wyniki są wtedy zwracane we właściwej kolejności.

### Co warto zapamiętać?

- Zdarzenia w aplikacjach w JavaScriptcie zachodzą niedeterministycznie (czyli w nieprzewidywalnej kolejności).
- Stosuj licznik, aby uniknąć zjawiska wyścigu do danych przy wykonywaniu współbieżnych operacji.

## Sposób 67. Nigdy nie uruchamiaj synchronicznie asynchronicznych wywołań zwrotnych

Wyobraź sobie wersję funkcji `downloadAsync` z pamięcią podręczną (zaimplementowaną jako obiekt typu `Dict`; zobacz sposób 45.), co pozwala uniknąć wielokrotnego pobierania tego samego pliku. Gdy wszystkie pliki znajdują się już w pamięci podręcznej, dobrym pomysłem może wydawać się natychmiastowe uruchomienie wywołania zwrotnego.

```

var cache = new Dict();

function downloadCachingAsync(url, onsuccess, onerror) {
  if (cache.has(url)) {
    onsuccess(cache.get(url)); // Wywołanie synchroniczne
    return;
  }
  return downloadAsync(url, function(file) {
    cache.set(url, file);
    onsuccess(file);
  }, onerror);
}

```

Choć udostępnianie danych natychmiast, gdy są już gotowe, wydaje się naturalne, narusza to oczekiwania klientów asynchronicznego interfejsu API. Przede wszystkim zmienia to oczekiwaną kolejność operacji. W sposobie 62. przedstawiony jest przytoczony poniżej przykład. Ten kod dla poprawnego asynchronicznego interfejsu API zawsze powinien rejestrować komunikaty w przewidywalnej kolejności.

```

downloadAsync("file.txt", function(file) {
  console.log("Zakończenie");
});
console.log("Rozpoczęcie");

```

Przy prostej, pokazanej wcześniej implementacji funkcji `downloadCachingAsync` przykładowy kod może rejestrować zdarzenia w dowolnej kolejności w zależności od tego, czy plik znajduje się w pamięci podręcznej.

```
downloadCachingAsync("file.txt", function(file) {  
    console.log("Zakończenie"); // Może zostać zapisane jako pierwsze  
});  
console.log("Rozpoczęcie");
```

Kolejność rejestrowanych komunikatów to tylko jedna sprawa. Ogólnie asynchroniczne interfejsy API służą do ścisłego rozdzielania przebiegów pętli zdarzeń. W sposobie 61. wyjaśniono, że upraszcza to obsługę współbieżności, ponieważ kod z jednego przebiegu pętli zdarzeń nie musi uwzględniać innego kodu, który jednocześnie zmienia współużytkowane struktury danych. Asynchroniczne wywołanie zwrotne uruchamiane synchronicznie narusza ten podział i powoduje, że kod przeznaczony do wykonania w odrębnym przebiegu pętli zdarzeń może zostać uruchomiony jeszcze w bieżącym przebiegu.

Załóżmy, że aplikacja przechowuje kolejkę plików pozostałych do pobrania i wyświetla użytkownikowi komunikat.

```
downloadCachingAsync(remaining[0], function(file) {  
    remaining.shift();  
    // ...  
});  
  
status.display("Trwa pobieranie " + remaining[0] + "...");
```

Jeśli wywołanie zwrotne jest uruchamiane synchronicznie, w wyświetlanym komunikacie znajdzie się niewłaściwa nazwa pliku (lub, co gorsza, wartość `"undefined"`, gdy kolejka jest pusta).

Uruchamianie asynchronicznych wywołań zwrotnych może prowadzić do jeszcze bardziej zawiłych problemów. W sposobie 64. wyjaśniono, że asynchroniczne wywołania zwrotne są przeznaczone do uruchamiania przy zwykle pustym stosie wywołań. Dlatego można bezpiecznie implementować asynchroniczne pętle za pomocą funkcji rekurencyjnych i nie grozi to przepełnieniem stosu wywołań. Synchroniczne wywołania sprawiają, że przestaje to być prawdą — pozornie asynchroniczna pętla może wtedy zająć całą pamięć stosu wywołań. Następnym problemem są wyjątki. Jeśli w pokazanej implementacji funkcji `downloadCachingAsync` wywołanie zwrotne spowoduje wyjątek, zostanie on zgłoszony w przebiegu pętli zdarzeń, który zainicjował proces pobierania, a nie (co byłoby zgodne z oczekiwaniami) w odrębnym przebiegu.

Aby mieć pewność, że wywołania zwrotne zawsze będą uruchamiane asynchronicznie, można wykorzystać istniejący asynchroniczny interfejs API. Podobnie jak w sposobach 65. i 66. wystarczy zastosować funkcję biblioteczną `setTimeout`, aby dodać wywołanie zwrotne do kolejki zdarzeń po minimalnym odstępie czasu. W niektórych platformach istnieją lepsze niż funkcja `setTimeout` rozwiązania do natychmiastowego dodawania zdarzeń do kolejki.

```
var cache = new Dict();

function downloadCachingAsync(url, onSuccess, onError) {
  if (cache.has(url)) {
    var cached = cache.get(url);
    setTimeout(onSuccess.bind(null, cached), 0);
    return;
  }
  return downloadAsync(url, function(file) {
    cache.set(url, file);
    onSuccess(file);
  }, onError);
}
```

W tej wersji wykorzystano metodę `bind` (zobacz sposób 25.) do zapisywania wyniku jako pierwszego argumentu wywołania zwrótnego `onSuccess`.

### Co warto zapamiętać?

- Nigdy nie uruchamiaj synchronicznie asynchronicznych wywołań zwrótnych — nawet jeśli dane są natychmiast dostępne.
- Synchroniczne uruchamianie asynchronicznych wywołań zwrótnych zakłóca oczekiwaną sekwencję operacji i może prowadzić do niespodziewanego przeplatania się instrukcji.
- Synchroniczne uruchamianie asynchronicznych wywołań zwrótnych może prowadzić do przepełnienia stosu i nieodpowiednio obsługiwanych wyjątków.
- Stosuj asynchroniczne interfejsy API (takie jak funkcja `setTimeout`) do planowania wykonania asynchronicznych wywołań zwrótnych w następnym przebiegu pętli zdarzeń.

## Sposób 68. Stosuj obietnice, aby zwiększyć przejrzystość asynchronicznego kodu

Popularnym sposobem pisania asynchronicznych interfejsów API jest stosowanie **obietnic** (ang. *promise*; inne nazwy to *deferreds* i *futures*). Asynchroniczne interfejsy API opisane w tym rozdziale przyjmują jako argumenty wywołania zwrótne.

```
downloadAsync("file.txt", function(file) {
  console.log("Plik: " + file);
});
```

Interfejs API oparty na obietnicach nie przyjmuje wywołań zwrótnych. Zamiast tego zwraca obiekt obietnicy, który sam przyjmuje wywołania zwrótne w metodzie `then`.

```
var p = downloadP("file.txt");

p.then(function(file) {
    console.log("Plik: " + file);
});
```

Na razie kod nie różni się w dużym stopniu od swej wcześniejszej wersji. Jednak wartość obietnic polega na *możliwości ich łączenia*. Wywołanie zwrotne przekazane do metody `then` można wykorzystać nie tylko do wywoływania określonych efektów (w przykładowym kodzie jest to wyświetlanie komunikatu w konsoli), ale też do generowania wyników. Dzięki zwróceniu wartości z wywołania zwrotnego można utworzyć nową obietnicę.

```
var fileP = downloadP("file.txt");

var lengthP = fileP.then(function(file) {
    return file.length;
});

lengthP.then(function(length) {
    console.log("Długość: " + length);
});
```

Obietnicę możesz traktować jak obiekt reprezentujący **ostateczną wartość**. Opakowuje on współbieżną operację, która jeszcze nie jest ukończona, ale ostatecznie zwróci wynikową wartość. Metoda `then` pozwala użyć jednej obietnicy reprezentującej określony typ ostatecznej wartości i wygenerować nową obietnicę odpowiadającą wartości innego typu (zwracanej przez wywołanie zwrotne).

Możliwość tworzenia nowych obietnic na podstawie istniejących zapewnia dużą swobodę i pozwala korzystać z prostych, ale dających duże możliwości idiomów. Można na przykład stosunkowo łatwo napisać narzędzie do złączania wyników z różnych obietnic.

```
var filesP = join(downloadP("file1.txt"),
                  downloadP("file2.txt"),
                  downloadP("file3.txt"));

filesP.then(function(files) {
    console.log("file1: " + files[0]);
    console.log("file2: " + files[1]);
    console.log("file3: " + files[2]);
});
```

Biblioteki obietnic często udostępniają funkcję narzędziową `when`, którą można stosować w podobny sposób.

```
var fileP1 = downloadP("file1.txt"),
    fileP2 = downloadP("file2.txt"),
    fileP3 = downloadP("file3.txt");

when([fileP1, fileP2, fileP3], function(files) {
    console.log("file1: " + files[0]);
});
```

```
console.log("file2: " + files[1]);
console.log("file3: " + files[2]);
});
```

Jedną z przyczyn, dla których obietnice zapewniają bardzo wygodny poziom abstrakcji, jest to, że przekazują wyniki za pomocą zwracania wartości przez metody `then` lub przez łączenie obietnic za pomocą narzędzi takich jak `join`. Nie muszą zapisywać informacji we współużytkowanych strukturach danych za pomocą współbieżnych wywołań zwrotnych. Model oparty na obietnicach jest z natury bezpieczniejszy, ponieważ pozwala uniknąć opisanego w sposobie 66. wyścigu do danych. Nawet najbardziej staranny programista może popełnić proste błędy przy zapisywaniu wyników asynchronicznych operacji we współużytkowanych zmiennych lub strukturach danych.

```
var file1, file2;

downloadAsync("file1.txt", function(file) {
    file1 = file;
});

downloadAsync("file2.txt", function(file) {
    file1 = file; // Niewłaściwa zmienna
});
```

Obietnice pozwalają uniknąć takich błędów, ponieważ łączenie obietnic nie wymaga modyfikowania współużytkowanych danych.

Zauważ też, że sekwencyjne łańcuchy asynchronicznych instrukcji przy stosowaniu obietnic są rzeczywiście zapisywane sekwencyjnie, a nie przy użyciu przedstawionego w sposobie 62. wzorca z nieczytelnym zagnieżdżaniem poleceń. Co więcej, obietnice pozwalają na automatyczne przekazywanie obsługi błędów. Gdy za pomocą obietnic łączysz w łańcuch kolekcję asynchronicznych operacji, możesz dla całej sekwencji utworzyć jedno wywołanie zwrotne do obsługi błędów, zamiast przekazywać je w każdym kroku (co było konieczne w kodzie w sposobie 63.).

Mimo to czasem przydatne jest celowe powodowanie sytuacji wyścigu. Obietnice zapewniają elegancki mechanizm, który to umożliwia. Możliwe, że aplikacja ma jednocześnie próbować wczytywać ten sam plik z kilku różnych serwerów i pobierać pierwszy ukończony plik. Narzędzie `select` (czasem używa się też nazwy `choose`) przyjmuje kilka obietnic i zwraca obietnicę z pierwszym dostępnym wynikiem. Oznacza to wyścig kilku obietnic.

```
var fileP = select(downloadP("http://example1.com/file.txt"),
    downloadP("http://example2.com/file.txt"),
    downloadP("http://example3.com/file.txt"));

fileP.then(function(file) {
    console.log("Plik: " + file);
});
```

Funkcję `select` można też wykorzystać do tworzenia limitów czasu w celu anulowania operacji, których wykonywanie trwa zbyt długo.

```
var fileP = select(downloadP("file.txt"), timeoutErrorP(2000));

fileP.then(function(file) {
    console.log("Plik: " + file);
}, function(error) {
    console.log("Błąd wejścia-wyjścia lub przekroczenie limitu czasu: " + error);
});
```

W tym przykładzie pokazano, że jako drugi argument metody `then` można przekazać do obietnicy wywołanie zwrotne do obsługi błędów.

### Co warto zapamiętać?

- Obietnice reprezentują ostateczne wartości (są współbieżnymi obliczeniami, które ostatecznie zwracają wynik).
- Stosuj obietnice do łączenia różnych operacji współbieżnych.
- Stosuj oparte na obietnicach interfejsy API, aby uniknąć wyścigu do danych.
- Stosuj funkcję `select` (stosowana bywa też nazwa `choose`) w sytuacjach, gdy celowo chcesz wywołać sytuację wyścigu.

# Skorowidz

## A

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

## B

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

## C

- currying, 87, 88

## D

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

## E

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

## F

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

**funkcja**

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

**funkcje**

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

**G****graf**

- sceny, 111
- sieci społecznościowej, 137

**H**

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

**I**

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

- var, 39

- with, 52, 53

- interfejs API, 151

**interfejsy**

- bezstanowe, 161
- stanowe, 161
- elastyczne, 164
- introspekcja, 119
- iterator, 84

**J**

- jawna konwersja, 36
- jednostki kodowe, 43

**K****klasa, 98**

- Array, 118, 146
- Dict, 124
- MWPage, 165
- User, 106

**kodowanie**

- o zmiennej długości, 45
- znaków, 43

**koercja, 171**

- kolejka zdarzeń, 179, 193

- kolekcja, 123

- kolekcje uporządkowane, 132

- konstrukcja try...catch, 59

- konstruktor, 71

- klasy Array, 148

- kontekst, 112

- konwencje spójne, 151

- konwersja typu, 28

- jawna, 36

- niejawna, 35

**L**

- liczba argumentów, 79

**liczby**

- o podwójnej precyzji, 26
- zmiennoprzecinkowe, 24

- literały tablicowe, 148

**Ł****łańcuch**

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



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

## M

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

## N

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

nieprzenośne określanie zasięgu, 62, 65  
niestandardowy odbiorca, 77

## O

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

## P

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

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

## R

referencje do obiektu arguments, 84  
 rekurencja, 190

## S

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

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

## Ś

śląd stosu, 93  
 średnik, 37  
 środowisko leksykalne, 52

## T

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

## U

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

## W

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

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

## Z

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

# PROGRAM PARTNERSKI

*GRUPY WYDAWNICZEJ HELION*



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

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

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