Najlepsze porady dla programistów JavaScriptu!

John Resig
Bear Bibeault

TAJEMNICE JAVASCRIPTU

PODRĘCZNIK NINJA

Tytuł oryginału: Secrets of the JavaScript Ninja

Tłumaczenie: Piotr Pilch

ISBN: 978-83-246-8504-2

Original edition copyright © 2013 by Manning Publications Co.
All rights reserved

Polish edition copyright © 2014 by HELION SA.
All rights reserved.

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

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

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

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

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

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

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

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

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa 9
Podziękowania 11
O książce 13
O autorach 19

CZĘŚĆ I. PRZYGOTOWANIE DO TRENINGU 21

Rozdział 1. Zostań wojownikiem 23

1.1. Omawiane biblioteki języka JavaScript 24
1.2. Język JavaScript 25
1.3. Kwestie dotyczące obsługi wielu przeglądarek 26
1.4. Najlepsze obecnie praktyki 30
1.4.1. Najlepsze obecnie praktyki — testowanie 30
1.4.2. Najlepsze obecnie praktyki — analizowanie wydajności 31
1.5. Podsumowanie 32

Rozdział 2. Broń w postaci testowania i debugowania 33

2.1. Debugowanie kodu 34
2.1.1. Rejestrowanie 34
2.1.2. Punkty wstrzymania 36
2.2. Generowanie testu 38
2.3. Środowiska testowania 40
2.3.1. QUnit 43
2.3.2. YUI Test 43
2.3.3. JsUnit 43
2.3.4. Nowsze środowiska testów jednostkowych 43
2.4. Fundamenty pakietu testów 44
2.4.1. Asercja 44
2.4.2. Grupy testów 45
2.4.3. Testowanie asynchroniczne 47
2.5. Podsumowanie 49

CZĘŚĆ II. TRENING UCZNIA 51

Rozdział 3. Funkcje są najważniejsze 53

3.1. Na czym polega funkcyjność? 54
3.1.1. Dlaczego ważna jest funkcyjna natura języka JavaScript? 55
3.1.2. Sortowanie za pomocą komparatora 60

Kup ksi k Pole ksi k

3.2. Deklaracje 63
3.2.1. Określanie zasięgu i funkcje 66
3.3. Wywołania 71
3.3.1. Od argumentów do parametrów funkcji 72
3.3.2. Wywołanie funkcji jako funkcji 73
3.3.3. Wywołanie funkcji jako metody 74
3.3.4. Wywołanie funkcji jako konstruktora 77
3.3.5. Wywołanie za pomocą metod apply() i call() 80
3.4. Podsumowanie 84

Rozdział 4. Posługiwanie się funkcjami 87

4.1. Funkcje anonimowe 88
4.2. Rekurencja 90
4.2.1. Rekurencja w funkcjach z nazwą 90
4.2.2. Rekurencja z metodami 92
4.2.3. Problem z podkradanym odwołaniem 93
4.2.4. Wstawiane funkcje z nazwą 95
4.2.5. Właściwość callee 97
4.3. Używanie funkcji jako obiektów 98
4.3.1. Przechowywanie funkcji 99
4.3.2. Funkcje z automatycznym zapamiętywaniem 100
4.3.3. Oszukiwanie metod tablicowych 103
4.4. Listy argumentów o zmiennej długości 105
4.4.1. Użycie metody apply() do dostarczania zmiennej argumentów 105
4.4.2. Przeciążanie funkcji 107
4.5. Sprawdzanie pod kątem funkcji 116
4.6. Podsumowanie 118

Rozdział 5. Zamknięcie się w domknięciach 119

5.1. Sposób działania domknięć 120
5.2. Praktyczne wykorzystanie domknięć 125
5.2.1. Zmienne prywatne 125
5.2.2. Wywołania zwrotne i liczniki czasu 127
5.3. Powiązanie kontekstów funkcji 131
5.4. Częściowe stosowanie funkcji 136
5.5. Przesłanianie działania funkcji 139
5.5.1. Zapamiętywanie 139
5.5.2. Opakowanie funkcji 142
5.6. Funkcje bezpośrednie 144
5.6.1. Zasięg tymczasowy i zmienne prywatne 146
5.6.2. Pętle 150
5.6.3. Opakowywanie biblioteki 152
5.7. Podsumowanie 153

Kup ksi k Pole ksi k

Rozdział 6. Obiektowość z prototypami 155

6.1. Tworzenie instancji i prototypy 156

6.1.1. Tworzenie instancji obiektu 156

6.1.2. Określanie typu obiektu za pośrednictwem konstruktorów 164

6.1.3. Dziedziczenie i łańcuch prototypów 166

6.1.4. Prototypy modelu DOM języka HTML 171

6.2. Pułapki! 173

6.2.1. Rozszerzanie obiektu 173

6.2.2. Rozszerzanie liczby 175

6.2.3. Używanie podklas dla wbudowanych obiektów 177

6.2.4. Problemy z tworzeniem instancji 178

6.3. Pisanie kodu bardziej zbliżonego do kodu z klasami 183

6.3.1. Sprawdzanie pod kątem serializacji funkcji 186

6.3.2. Inicjalizacja podklas 187

6.3.3. Zachowywanie supermetod 188

6.4. Podsumowanie 190

Rozdział 7. Borykanie się z wyrażeniami regularnymi 193

7.1. Dlaczego wyrażenia regularne są tak ważne? 194

7.2. Odświeżenie informacji o wyrażeniach regularnych 195

7.2.1. Omówienie wyrażeń regularnych 195

7.2.2. Wyrazy i operatory 197

7.3. Kompilowanie wyrażeń regularnych 201

7.4. Przechwytywanie pasujących segmentów 204

7.4.1. Wykonywanie prostych przechwytywań 204

7.4.2. Dopasowywanie za pomocą globalnych wyrażeń regularnych 205

7.4.3. Przywoływanie przechwytywań 207

7.4.4. Grupy bez przechwytywania 208

7.5. Zastępowanie za pomocą funkcji 209

7.6. Rozwiązywanie typowych problemów z wykorzystaniem wyrażeń regularnych 212

7.6.1. Obcinanie łańcucha 212

7.6.2. Dopasowywanie znaków nowego wiersza 214

7.6.3. Unicode 215

7.6.4. Znaki o zmienionym znaczeniu 216

7.7. Podsumowanie 217

Rozdział 8. Wątki i liczniki czasu 219

8.1. Sposób działania liczników czasu i wątkowości 220

8.1.1. Ustawianie i usuwanie liczników czasu 220

8.1.2. Wykonywanie licznika czasu w obrębie wątku wykonywania 221

8.1.3. Różnice między czasami oczekiwania i interwałami 223

8.2. Minimalne opóźnienie licznika czasu i wiarygodność 225

8.3. Radzenie sobie z przetwarzaniem kosztownym obliczeniowo 228

8.4. Scentralizowane kontrolowanie liczników czasu 231

8.5. Testowanie asynchroniczne 235

8.6. Podsumowanie 236

Kup ksi k Pole ksi k

CZĘŚĆ III. TRENING WOJOWNIKA 237

Rozdział 9. Alchemia wojownika. Analizowanie kodu w środowisku wykonawczym 239

9.1. Mechanizmy analizy kodu 240
9.1.1. Analizowanie za pomocą metody eval() 240
9.1.2. Analizowanie za pośrednictwem konstruktora Function 243
9.1.3. Analizowanie przy użyciu liczników czasu 244
9.1.4. Analizowanie w zasięgu globalnym 244
9.1.5. Bezpieczne analizowanie kodu 247
9.2. „Dekompilacja” funkcji 248
9.3. Analizowanie kodu w praktyce 251
9.3.1. Przekształcanie łańcuchów JSON 251
9.3.2. Importowanie kodu z przestrzenią nazw 253
9.3.3. Kompresja i ukrywanie kodu JavaScript 254
9.3.4. Dynamiczne przebudowywanie kodu 256
9.3.5. Znaczniki skryptu zorientowanego aspektowo 257
9.3.6. Metajęzyki i języki DSL 258
9.4. Podsumowanie 262

Rozdział 10. Instrukcje with 263

10.1. O co chodzi z instrukcją with? 264
10.1.1. Przywoływanie właściwości w zasięgu instrukcji with 264
10.1.2. Przypisania w zasięgu instrukcji with 266
10.1.3. Kwestie dotyczące wydajności 268
10.2. Rzeczywiste przykłady 270
10.3. Importowanie kodu z przestrzenią nazw 272
10.4. Testowanie 272
10.5. Stosowanie szablonów z instrukcją with 273
10.6. Podsumowanie 276

Rozdział 11. Opracowywanie strategii obsługi wielu przeglądarek 277

11.1. Wybór przeglądarek do obsługi 278
11.2. Pięć podstawowych kwestii programistycznych 279
11.2.1. Błędy i różnice w przeglądarkach 281
11.2.2. Poprawki błędów w przeglądarce 281
11.2.3. Radzenie sobie z zewnętrznym kodem i znacznikami 283
11.2.4. Brakujące funkcje 289
11.2.5. Regresje 290
11.3. Strategie implementowania 292
11.3.1. Bezpieczne poprawki dla różnych przeglądarek 292
11.3.2. Wykrywanie obiektu 294
11.3.3. Symulacja funkcji 295
11.4. Zmniejszanie liczby założeń 301
11.5. Podsumowanie 303

Kup ksi k Pole ksi k

Rozdział 12. Atrybuty, właściwości i arkusze stylów CSS 305

12.1. Atrybuty i właściwości modelu DOM 307
12.1.1. Nazewnictwo w różnych przeglądarkach 308
12.1.2. Ograniczenia dotyczące nazw 309
12.1.3. Różnice między językami XML i HTML 310
12.1.4. Działanie atrybutów niestandardowych 310
12.1.5. Kwestie dotyczące wydajności 311
12.2. Problemy z atrybutami w przypadku obsługi wielu przeglądarek 315
12.2.1. Rozszerzanie nazwy (identyfikatora) modelu DOM 315
12.2.2. Normalizacja adresu URL 317
12.2.3. Atrybut style 318
12.2.4. Atrybut type 319
12.2.5. Problem z indeksem tabulacji 320
12.2.6. Nazwy węzłów 321
12.3. Problemy związane z atrybutami stylów 321
12.3.1. Gdzie są moje style? 322
12.3.2. Określanie nazw właściwości stylów 324
12.3.3. Właściwość stylów float 326
12.3.4. Konwersja wartości pikseli 326
12.3.5. Określanie wysokości i szerokości 327
12.3.6. Przenikanie nieprzezroczystości 332
12.3.7. Poskromienie kolorowego koła 335
12.4. Uzyskiwanie stylów obliczanych 338
12.5. Podsumowanie 341

CZĘŚĆ IV. TRENING MISTRZA 343

Rozdział 13. Radzenie sobie ze zdarzeniami 345

13.1. Techniki wiązania zdarzeń i anulowania powiązań 346
13.2. Obiekt Event 351
13.3. Zarządzanie procedurami obsługi 355
13.3.1. Scentralizowane przechowywanie powiązanych informacji 355
13.3.2. Zarządzanie procedurami obsługi zdarzeń 358
13.4. Wyzwalanie zdarzeń 369
13.4.1. Zdarzenia niestandardowe 371
13.5. Propagacja i delegowanie 375
13.5.1. Delegowanie zdarzeń do elementu nadrzędnego 376
13.5.2. Radzenie sobie z mankamentami przeglądarek 377
13.6. Zdarzenie gotowości dokumentu 387
13.7. Podsumowanie 389

Rozdział 14. Modyfikowanie modelu DOM 393

14.1. Umieszczanie kodu HTML w modelu DOM 394
14.1.1. Przekształcanie kodu HTML w model DOM 396
14.1.2. Wstawianie do dokumentu 399
14.1.3. Wykonywanie skryptu 401

Kup ksi k Pole ksi k

14.2. Klonowanie elementów 403
14.3. Usuwanie elementów 405
14.4. Treść tekstowa 407
14.4.1. Ustawianie tekstu 408
14.4.2. Pobieranie tekstu 409
14.5. Podsumowanie 410

Rozdział 15. Mechanizmy selektorów CSS 411

15.1. Interfejs API selektorów organizacji W3C 413
15.2. Użycie języka XPath do znajdowania elementów 416
15.3. Implementacja czystego modelu DOM 418
15.3.1. Analizowanie selektora 421
15.3.2. Znajdowanie elementów 422
15.3.3. Filtrowanie zestawu 423
15.3.4. Rekurencja i scalanie 424
15.3.5. Wstępujący mechanizm selektorów 425
15.4. Podsumowanie 427

Skorowidz 428

Kup ksi k Pole ksi k

12 Atrybuty, właściwości i arkusze stylów CSS

W tym rozdziale:

- Atrybuty i właściwości modelu DOM
- Radzenie sobie z atrybutami i stylami w wielu przeglądarkach
- Obsługa właściwości wymiarów elementu
- Wykrywanie stylów obliczanych

Z wyjątkiem poprzedniego rozdziału spora część książki została dotychczas poświęcona językowi JavaScript. Choć z samym językiem JavaScript związanych jest mnóstwo niuansów, po połączeniu go z modelem DOM przeglądarki wszystko stanie się naprawdę zagmatwane.

Zrozumienie pojęć związanych z modelem DOM, a także tego, jaki ma to związek z językiem JavaScript, stanowi kluczowy element na drodze do zostania wojownikiem języka JavaScript. W szczególności trzeba wspomnieć o zaskakujących sposobach, przy których użyciu pewne pojęcia dotyczące modelu DOM wydają się przeczyć logice. Atrybuty i właściwości modelu DOM wywołały u wielu twórców stron z kodem JavaScript poczucie zagubienia. W przypadku atrybutów i właściwości nie tylko występuje kilka bardzo specyficznych zachowań, ale również istnieją kwestie, którym towarzyszy więcej błędów i problemów z obsługą wielu przeglądarek.

Kup ksi k Pole ksi k

Jednakże atrybuty i właściwości stanowią ważne zagadnienia. Atrybuty są integralną częścią procesu budowania modelu DOM, a właściwości zapewniają podstawowe środki przechowywania przez elementy informacji o środowisku wykonawczym, a ponadto uzyskiwania do nich dostępu.

Przyjrzyjmy się krótkiemu przykładowi, który demonstruje możliwości wprowadzania w stan konsternacji:

```
<img src="../images/ninja-with-nunchuks.png">
<script type="text/javascript">
  var image = document.getElementsByTagName('img')[0];
  var newSrc = '../images/ninja-with-pole.png';
  image.src = newSrc;
  assert(image.src === newSrc,
         'Źródło obrazu to teraz ' + image.src);
  assert(image.getAttribute('src') === '../images/ninja-with-nunchuks.png',
         'Atrybut src obrazu to ' + image.getAttribute('src'));
</script>
```

W tym fragmencie kodu tworzony jest znacznik obrazu `image`, uzyskiwane jest odwołanie do niego i zmieniana jest wartość właściwości `src` na nową. Choć wydaje się to naprawdę proste, dla pewności uruchamiane są następujące dwa testy:

- Sprawdzane jest, czy właściwość `src` uzyskała wartość, którą właśnie jej przekazano W końcu, jeśli zostanie zdefiniowane przypisanie `x = 213`, z pewnością będzie można się spodziewać, że wartością zmiennej `x` będzie `213`.
- Nie zmodyfikowano atrybutu, dlatego powinien pozostać bez zmian. Czy to prawda?

Jednakże po załadowaniu kodu w przeglądarce okaże się, że oba testy się nie powiodą.

Stwierdzimy, że właściwość `src` nie ma przypisanej wartości, lecz raczej coś podobnego do:

```
http://localhost/ninja/images/ninja-with-pole.png
```

Czy jeśli właściwości przypisano wartość, nie należy spodziewać się, że będzie mieć dokładnie taką wartość?

Co jeszcze dziwniejsze, choć nie zmodyfikowano atrybutu w elemencie, niepowodzenie testu pokazuje, że wartość *atrybutu* `src` zmieniła się na:

```
../images/ninja-with-pole.png
```

O co w tym chodzi?

W rozdziale zostaną omówione wszystkie zagadki stwarzane przez przeglądarki w odniesieniu do właściwości i atrybutów elementów. Ponadto wyjaśnimy, dlaczego wyniki nie były dokładnie takie, jakich oczekiwano.

To samo dotyczy arkuszy stylów CSS i określania stylów elementów. Wiele trudności, które pojawiają się podczas tworzenia dynamicznej aplikacji internetowej, wynika ze złożoności ustawiania i pobierania stylów elementów. Choć w książce nie

Kup ksi k Pole ksi k

będzie możliwe pomieszczenie całej wiedzy na temat obsługi stylów elementów (jest ona wystarczająco obszerna, by mogła zapełnić całą osobną książkę), zostaną omówione najważniejsze rzeczy.

Zacznijmy od dokładnego zrozumienia, czym są atrybuty i właściwości elementów.

12.1. *Atrybuty i właściwości modelu DOM*

W przypadku uzyskiwania dostępu do wartości atrybutów elementów możliwe są dwie następujące opcje: użycie tradycyjnych metod `getAttribute` i `setAttribute` modelu DOM lub zastosowanie właściwości obiektów modelu DOM, które odpowiadają atrybutom.

Aby na przykład uzyskać atrybut `id` elementu, którego odwołanie jest przechowywane w zmiennej `e`, możesz użyć jednej z następujących instrukcji:

```
e.getAttribute('id')
e.id
```

W każdym przypadku zostanie uzyskana wartość atrybutu `id`.

Przeanalizujmy poniższy kod (listing 12.1), aby lepiej zrozumieć, jak zachowują się wartości atrybutów oraz odpowiadające im właściwości.

Listing 12.1. Uzyskiwanie dostępu do wartości atrybutów za pośrednictwem metod i właściwości modelu DOM

```
<div></div>

<script type="text/javascript">

  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];
    div.setAttribute("id","ninja-1");
    assert(div.getAttribute('id') === "ninja-1",
           "Pomyślnie zmieniono atrybut.");

    div.id = "ninja-2";
    assert(div.id === "ninja-2",
           "Pomyślnie zmieniono właściwość.");

    div.id = "ninja-3";
    assert(div.id === "ninja-3",
           "Pomyślnie zmieniono właściwość.");
    assert(div.getAttribute('id') === "ninja-3",
           "Pomyślnie zmieniono atrybut za pośrednictwem właściwości.");

    div.setAttribute("id","ninja-4");
    assert(div.id === "ninja-4",
           "Pomyślnie zmieniono właściwość za pośrednictwem atrybutu.");
    assert(div.getAttribute('id') === "ninja-4",
           "Pomyślnie zmieniono atrybut.");

  };

</script>
```

❶ **Uzyskuje odwołanie do elementu.**

❷ **Testuje metodę modelu DOM.**

❸ **Testuje wartość właściwości.**

❹ **Testuje zgodność właściwości (atrybutu).**

❺ **Dodatkowo testuje zgodność właściwości (atrybutu).**

W przykładzie zaprezentowano interesujące zachowanie dotyczące atrybutów i właściwości elementów. Najpierw definiowany jest prosty element `<div>`, który będzie używany jako przedmiot testu. W obrębie procedury obsługi ładowania strony (zapewnia, że zostało zakończone budowanie modelu DOM) uzyskiwane jest odwołanie do jedynego elementu `<div>` ❶, a następnie wykonywanych jest kilka testów.

W pierwszym teście ❷ dla atrybutu `id` ustawiono wartość `"ninja-1"` za pośrednictwem metody `setAttribute()`. Następnie potwierdzane jest, że metoda `getAttri↪bute()` zwraca tę samą wartość dla tego samego atrybutu. Nie powinno być zaskoczeniem stwierdzenie, że ten test zadziała po prostu świetnie po załadowaniu strony.

Podobnie w następnym teście ❸ dla właściwości `id` ustawiana jest wartość `"ninja-2"`, a następnie sprawdzane jest, czy wartość właściwości faktycznie została zmieniona. Żaden problem.

Przy następnym teście ❹ wszystko zaczyna się robić interesujące. Ponownie dla właściwości `id` ustawiana jest nowa wartość (w tym przypadku `"ninja-3"`), po czym jeszcze raz sprawdzane jest, czy zmieniła się wartość właściwości. Tym razem jednak potwierdzane jest także, że nie tylko powinna zostać zmieniona wartość właściwości, ale również wartość *atrybutu* `id`. Obie asercje kończą się powodzeniem. Na podstawie tego stwierdzamy, że właściwość `id` i atrybut `id` są jakoś ze sobą powiązane. Zmiana właściwości `id` spowoduje też zmodyfikowanie wartości atrybutu `id`.

Następny test ❺ potwierdza, że sprawdza się także inne rozwiązanie: ustawienie wartości atrybutu również powoduje zmianę odpowiedniej wartości właściwości.

Nie pozwól jednak wprowadzić się tym w błąd, myśląc, że właściwość i atrybut używają tej samej wartości, ponieważ tak nie jest. W dalszej części rozdziału okaże się, że atrybut i odpowiadająca mu właściwość, choć powiązane, nie zawsze są identyczne. Była już o tym mowa na początku rozdziału.

W odniesieniu do atrybutów i właściwości istnieje pięć następujących ważnych kwestii do rozważenia:

- nazewnictwo w przypadku wielu przeglądarek,
- ograniczenia dotyczące nazewnictwa,
- różnice między językami HTML i XML,
- niestandardowe działanie atrybutów,
- wydajność.

Przyjrzyjmy się każdej z tych kwestii.

12.1.1. Nazewnictwo w różnych przeglądarkach

W przypadku określania nazw atrybutów i odpowiadających im właściwości nazwy właściwości są generalnie bardziej spójne w poszczególnych przeglądarkach. Jeśli możliwe jest uzyskanie dostępu w jednej przeglądarce do właściwości przy użyciu określonej nazwy, są spore szanse na to, że w innych przeglądarkach będzie stosowana taka sama nazwa. Choć istnieją *pewne* różnice, zwykle więcej różnic występuje w przypadku nazewnictwa atrybutów niż właściwości.

Kup książkę Poleć książkę

Choć na przykład w większości przeglądarek atrybut `class` można uzyskać jako `class`, przeglądarka Internet Explorer wymaga nazwy `className`. Prawdopodobnie wynika to z tego (jak się wkrótce okaże), że nazwa właściwości to `className`. Oznacza to, że w przeglądarce Internet Explorer nazwa atrybutu i nazwa właściwości są spójne. Choć spójność przeważnie jest czymś dobrym, różnice w nazewnictwie w przypadku poszczególnych przeglądarek mogą być dość frustrujące.

Biblioteki, takie jak jQuery, ułatwiają normalizację takich rozbieżności nazewniczych, umożliwiając określenie jednej nazwy niezależnie od platformy, a następnie przeprowadzenie w tle wszelkich wymaganych translacji. Jednakże bez wsparcia biblioteki konieczne będzie poznanie różnic i odpowiednie napisanie własnego kodu.

12.1.2. Ograniczenia dotyczące nazw

Nazwy atrybutów, które są reprezentowane przez łańcuchy przekazywane do metod modelu DOM, mogą być określane raczej dość swobodnie. Z kolei nazwy właściwości, które mogą być przywoływane jako identyfikatory z wykorzystaniem notacji z operatorem kropki, są bardziej ograniczone, ponieważ muszą być zgodne z regułami dotyczącymi identyfikatorów, a ponadto występuje kilka niedozwolonych słów zastrzeżonych.

W specyfikacji ECMAScript (dostępna pod adresem *http://www.ecma-international.org/publications/standards/Ecma-262.htm*) stwierdzono, że słowa kluczowe nie mogą być używane jako nazwy właściwości, dlatego zostały zdefiniowane alternatywy. Na przykład atrybut `for` elementów `<label>` jest reprezentowany przez właściwość `htmlFor`, ponieważ ten atrybut jest słowem zastrzeżonym, a atrybut `class` wszystkich elementów jest reprezentowany przez właściwość `className`, gdyż nazwa `class` również jest zastrzeżona. Dodatkowo nazwy atrybutów złożone z wielu słów, takie jak `readonly`, są reprezentowane przez nazwy właściwości z literami o zmiennej wielkości (w tym przypadku `readOnly`). W tabeli 12.1 zebrano więcej przykładów takich różnic.

Tabela 12.1. Przypadki, w których nazwy właściwości i atrybutów mogą się różnić

Nazwa atrybutu	Nazwa właściwości
`for`	`htmlFor`
`class`	`className`
`readonly`	`readOnly`
`maxlength`	`maxLength`
`cellspacing`	`cellSpacing`
`rowspan`	`rowSpan`
`colspan`	`colSpan`
`tabindex`	`tabIndex`
`cellpadding`	`cellPadding`
`usemap`	`useMap`
`frameborder`	`frameBorder`
`contenteditable`	`contentEditable`

Zauważ, że w standardzie HTML5 dodano nowe elementy i atrybuty, które po jakimś czasie będą wymagać dołączenia do tej listy. Są to między innymi następujące obiekty: `accessKey`, `contextMenu`, `dropZone`, `spellCheck`, `hrefLang`, `dateTime`, `pubDate`, `isMap`, `srcDoc`, `mediaGroup`, `autoComplete`, `noValidate` i `radioGroup`.

12.1.3. Różnice między językami XML i HTML

Cały pomysł na to, by właściwości automatycznie odpowiadały atrybutom, jest charakterystyczny dla modelu DOM języka HTML. W przypadku korzystania z modelu DOM języka XML żadna właściwość nie jest automatycznie tworzona w elementach w celu reprezentowania wartości atrybutów. A zatem do uzyskania wartości atrybutów konieczne będzie użycie tradycyjnych metod atrybutów modelu DOM. Nie jest to zbyt przerażające ograniczenie, ponieważ dokumenty XML zwykle nie wiążą się z normalną litanią pomyłek nazewniczych, które są obecne w przypadku atrybutów modelu DOM w dokumentach HTML.

UWAGA Termin *model DOM języka XML* oznacza jedynie strukturę obiektową tworzoną w pamięci do reprezentowania dokumentu XML w taki sam sposób, w jaki model DOM języka HTML reprezentuje dokument HTML.

Dobrym pomysłem jest zastosowanie w kodzie określonej formy sprawdzenia w celu określenia na potrzeby odpowiedniego kontynuowania działań, czy element (lub dokument) to element XML (lub dokument). Następująca funkcja stanowi przykład tego typu sprawdzenia:

```
function isXML(elem) {
  return (elem.ownerDocument ||
    elem.documentElement.nodeName.toLowerCase() !== "html";
}
```

Ta funkcja zwróci wartość `true`, jeśli element to element XML. W przeciwnym razie zostanie zwrócona wartość `false`.

12.1.4. Działanie atrybutów niestandardowych

Nie wszystkie atrybuty są reprezentowane przez właściwości elementów. Choć przeważnie jest to prawdą w przypadku atrybutów określanych przez model DOM języka HTML, *atrybuty niestandardowe*, które mogą zostać umieszczone w elementach na utworzonych stronach, nie są automatycznie reprezentowane przez właściwości elementu. Aby uzyskać dostęp do wartości atrybutu niestandardowego, konieczne jest użycie metod `getAttribute()` i `setAttribute()` modelu DOM.

Jeśli nie masz pewności, czy istnieje właściwość dla atrybutu, zawsze możesz wykonać pod tym kątem test, a następnie skorzystać z metod modelu DOM, gdy właściwości nie ma. Oto przykład:

```
var value = element.someValue ? element.someValue :
                                element.getAttribute('someValue');
```

Kup ksi k Pole ksi k

WSKAZÓWKA W przypadku standardu HTML5 użycie przedrostka `data-` dla wszystkich atrybutów niestandardowych powoduje, że pozostają one zgodne ze specyfikacją HTML5. Zalecane jest postąpienie w ten sposób nawet wtedy, gdy nadal używany jest standard HTML4, aby kod ze znacznikami przygotować z uwzględnieniem przyszłych rozwiązań. Poza tym jest to odpowiednia konwencja, która pozwala wyraźnie oddzielić atrybuty niestandardowe od wbudowanych.

12.1.5. Kwestie dotyczące wydajności

Ogólnie dostęp do właściwości jest szybszy niż dostęp do odpowiadających im metodom atrybutów modelu DOM, zwłaszcza w przeglądarce Internet Explorer. Przekonajmy się o tym.

Czy przypominasz sobie omówienie testowania wydajności z rozdziału 2.? Polega to na pomiarze czasu, jaki zajmie wielokrotne powtórzenie operacji. Nie można zmierzyć wydajności pojedynczej operacji, ponieważ czas jej trwania jest zbyt krótki, aby uzyskać dokładne wyniki (powróć do omówienia liczników czasu z rozdziału 8.).

Jeśli jedna operacja trwa zbyt krótko, aby przeprowadzić dla niej pomiar, co będzie w przypadku pięciu milionów operacji? Dokładnie taki pomiar przeprowadza listing 12.2.

Listing 12.2. Porównanie wydajności metod modelu DOM z właściwościami

```
<div id="testSubject"></div>

<script type="text/javascript">

  var count = 5000000;
  var n;
  var begin = new Date();
  var end;
  var testSubject = document.getElementById('testSubject');
  var value;

  for (n = 0; n < count; n++) {
    value = testSubject.getAttribute('id');
  }
  end = new Date();
  assert(true,'Czas odczytu metody modelu DOM: ' +
      (end.getTime() - begin.getTime()));

  begin = new Date();
  for (n = 0; n < count; n++) {
    value = testSubject.id;
  }
  end = new Date();
  assert(true,'Czas odczytu właściwości: ' +
         (end.getTime() - begin.getTime()));

  begin = new Date();
  for (n = 0; n < count; n++) {
```

Definiuje wcześniej zmienne.

Testuje odczyt metody modelu DOM.

Testuje odczyt właściwości.

Testuje zapis metody modelu DOM.

```
    testSubject.setAttribute('id','testSubject');
  }
  end = new Date();
  assert(true,'Czas zapisu metody modelu DOM: ' +
         (end.getTime() - begin.getTime()));

  begin = new Date();                              ← Testuje zapis właściwości.
  for (n = 0; n < count; n++) {
    testSubject.id = 'testSubject';
  }
  end = new Date();
  assert(true,'Czas zapisu właściwości: ' +
         (end.getTime() - begin.getTime()));

</script>
```

Ten kod przeprowadza test wydajności metod `getAttribute()` i `setAttribute()` modelu DOM dla podobnych operacji odczytu i zapisu odpowiedniej właściwości.

Test został uruchomiony w wielu przeglądarkach, a zebrane wyniki zaprezentowano w tabeli 12.2. Wszystkie czasy trwania podano w milisekundach. Jak widać, operacje uzyskiwania i ustawiania właściwości są niemal zawsze szybsze niż metody `getAttribute()` i `setAttribute()`.

UWAGA Większość tych testów przeprowadzono na komputerze MacBook Pro z 2011 roku, który zawierał procesor i7 o częstotliwości 2,8 GHz, 8 GB pamięci RAM i system OS X Lion. Testy w przeglądarce Internet Explorer zostały wykonane na komputerze PC z takim samym procesorem, 4 GB pamięci RAM i systemem Windows 7 (64-bitowym).

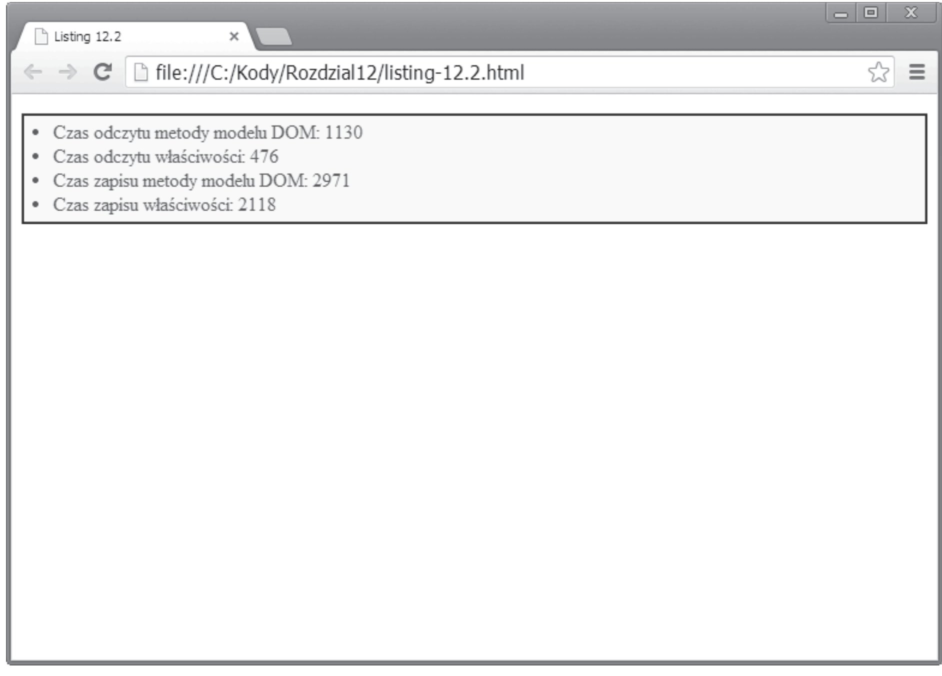
Tabela 12.2. **Wyniki testu wydajności porównującego czas dostępu do metod modelu DOM i właściwości**

Przeglądarka	getAttribute()	Właściwość get	setAttribute()	Właściwość set
Internet Explorer 9	3970	940	7667	956
Firefox 14	827	434	1414	1584
Safari 5	268	142	1055	627
Chrome 29	1130	476	2971	2118
Opera 12	2109	1642	2370	1635

Wyniki przykładowego uruchomienia tego testu pokazano na rysunku 12.1.

Choć podane różnice szybkości mogą nie robić wrażenia w przypadku pojedynczych operacji, mogą one się sumować przy wielu uruchomieniach testu (na przykład w ramach intensywnej pętli). Aby zwiększyć wydajność, można zaimplementować metodę, która umożliwia uzyskanie dostępu do wartości za pomocą właściwości (jeśli istnieje) lub metody modelu DOM (awaryjne rozwiązanie w przypadku braku właściwości). Przeanalizuj listing 12.3.

Kup ksi k Pole ksi k



Rysunek 12.1. Wyniki uruchomienia testu wydajności w przeglądarce Chrome

Listing 12.3. Funkcja służąca do ustawiania i pobierania wartości atrybutów

```
<div id="testSubject"></div>

<script type="text/javascript">

  (function(){                                          ❶ Tworzy zasięg prywatny.

    var translations = {                                ❷ Tworzy mapę translacji.
      "for": "htmlFor",
      "class": "className",
      readonly: "readOnly",
      maxlength: "maxLength",
      cellspacing: "cellSpacing",
      rowspan: "rowSpan",
      colspan: "colSpan",
      tabindex: "tabIndex",
      cellpadding: "cellPadding",
      usemap: "useMap",
      frameborder: "frameBorder",
      contenteditable: "contentEditable"
    };
                                                        ❸ Definiuje funkcję ustawiania
    window.attr = function(element,name,value) {          (pobierania).
      var property = translations[name] || name,
          propertyExists = typeof element[property] !== "undefined";

      if (typeof value !== "undefined") {
        if (propertyExists) {
```

```
        element[property] = value;
      }
      else {
        element.setAttribute(name,value);
      }
    }

    return propertyExists ?
      element[property] :
      element.getAttribute(name);
  };
})();

var subject = document.getElementById('testSubject');        ◄──── Testuje nową funkcję.
assert(attr(subject,'id') === 'testSubject',
       "Pobrano wartość atrybutu id.");

assert(attr(subject,'id','other') === 'other',
     "Ustawiono nową wartość atrybutu id.");
assert(attr(subject,'id') === 'other',
     "Pobrano nową wartość atrybutu.");

assert(attr(subject,'data-custom','whatever') === 'whatever',
       "Ustawiono atrybut niestandardowy.");
assert(attr(subject,'data-custom') === 'whatever',
       "Pobrano atrybut niestandardowy.");

</script>
```

W tym przykładzie nie tylko zdefiniowano funkcję ustawiającą oraz pobierającą dla wartości atrybutów i właściwości, ale też zaprezentowano kilka ważnych pojęć, które mogą zostać wykorzystane w dowolnym miejscu kodu.

W przykładowej funkcji konieczne jest przeprowadzenie translacji między nazwami właściwości i atrybutów (zgodnie z opisem w tabeli 12.1), dlatego utworzono mapę translacji ❷. Jednakże niewskazane jest zanieczyszczanie tą mapą globalnej przestrzeni nazw. Mapa ma być dostępna dla funkcji w jej zasięgu lokalnym, lecz nigdzie indziej.

W tym celu definicja mapy i deklaracja funkcji są umieszczane w obrębie funkcji bezpośredniej ❶, która tworzy zasięg lokalny. Mapa translacji ❷ nie jest dostępna poza obrębem funkcji bezpośredniej, ale funkcja ustawiająca (pobierająca), która również została zdefiniowana ❸ wewnątrz funkcji bezpośredniej, ma dostęp do mapy za pośrednictwem swojego domknięcia. Sprytne, prawda?

Kolejna ważna zasada jest prezentowana przez samą funkcję `attr()`, która może odgrywać zarówno rolę ustawiającej, jak i pobierającej po prostu przez sprawdzanie własnej listy argumentów. Jeśli funkcji przekazano argument `value`, będzie ona odgrywać rolę ustawiającej, ustawiając przekazaną wartość jako wartość atrybutu. Jeśli argument `value` zostanie pominięty, a przekazane zostaną tylko pierwsze dwa argumenty, funkcja odgrywa rolę pobierającej, uzyskując wartość konkretnego atrybutu.

W obu przypadkach zwracana jest wartość atrybutu, co ułatwia użycie funkcji w dowolnym z jej trybów w łańcuchu wywołań funkcji.

Kup ksi k Pole ksi k

Godne uwagi jest to, że powyższa implementacja nie uwzględnia wielu problemów dotyczących obsługi wielu przeglądarek, które towarzyszą dostępowi do atrybutów. Dowiedzmy się, jakie dokładnie są to problemy.

12.2. Problemy z atrybutami w przypadku obsługi wielu przeglądarek

Ogólnie problemy dotyczące obsługi wielu przeglądarek mogą być dość przerażające. Liczba takich problemów, które dotyczą wartości atrybutów, nie jest mała. Przeanalizujmy kilka podstawowych i najczęściej występujących problemów, rozpoczynając od rozszerzania nazwy modelu DOM.

12.2.1. Rozszerzanie nazwy (identyfikatora) modelu DOM

Najgorszym błędem, z jakim trzeba się uporać, jest niewłaściwa implementacja kodu modelu DOM w przeglądarkach.

Jak wskazano w poprzednim rozdziale, problem polega na tym, że wszystkie przeglądarki z Wielkiej Piątki pobierają wartości atrybutów `id` lub `name` określone w elementach wejściowych formularza i dodają odwołania do elementów jako właściwości elementu nadrzędnego `<form>`. Takie wygenerowane właściwości aktywnie przesłaniają wszelkie istniejące właściwości o tej samej nazwie, które mogą znajdować się już w elemencie formularza.

Ponadto przeglądarka Internet Explorer zastępuje nie tylko właściwości, lecz także wartości atrybutów odwołaniami do elementów.

Listing 12.4 demonstruje te problemy.

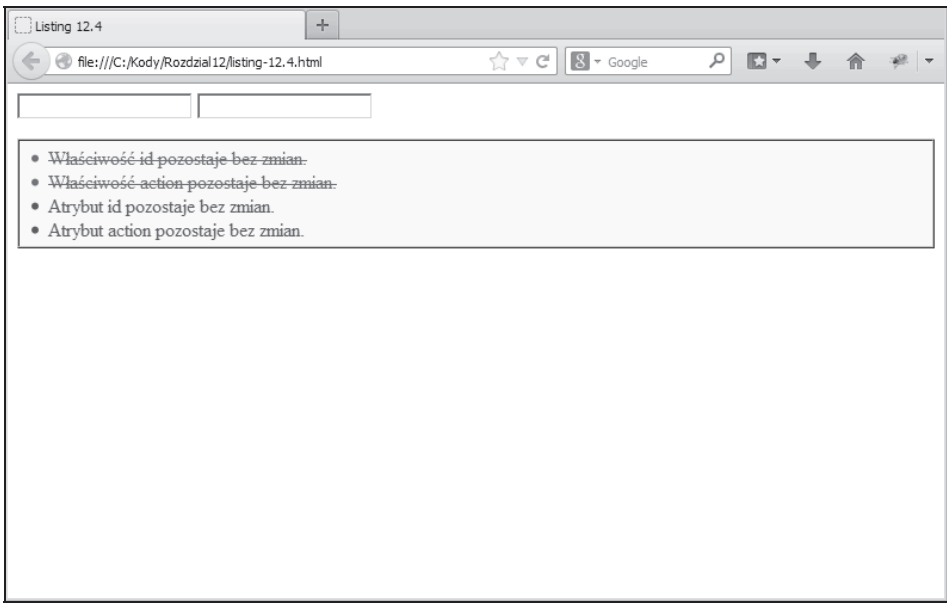
Listing 12.4. Demonstracja sposobu wymuszania przez przeglądarki swoich reguł dla elementów formularza

```
<form id="testForm" action="/">                         ❶ Tworzy przedmiot testu.
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>

<script type="text/javascript">
  window.onload = function(){

    var form = document.getElementById('testForm');
                                                        ❷ Testuje, czy właściwości zostały
    assert(form.id === 'testForm',                        pozostawione bez zmian.
           "Właściwość id pozostaje bez zmian.");
    assert(form.action === '/',
           "Właściwość action pozostaje bez zmian.");
                                                        ❸ Testuje, czy atrybuty zostały
    assert(form.getAttribute('id') === 'testForm',        zmodyfikowane.
           "Atrybut id pozostaje bez zmian.");
    assert(form.getAttribute('action') === '/',
           "Atrybut action pozostaje bez zmian.");

  };
</script>
```

Ten zestaw testów pokazuje, jak ta niefortunna cecha może spowodować utratę danych znaczników. Najpierw definiowany jest formularz HTML ❶ z dwoma podrzędnymi elementami wejściowymi. Pierwszy z nich ma identyfikator `id`, a drugi nazwę `action`.

Pierwszy test potwierdza ❷, że właściwości `id` i `action` elementu formularza powinny mieć postać określoną w znacznikach HTML. Z kolei drugi zestaw testów ❸ potwierdza, że wartości atrybutów odzwierciedlają znaczniki.

Jednakże po uruchomieniu testu w przeglądarce Chrome zostanie uzyskany wynik widoczny na rysunku 12.2.



Rysunek 12.2. Wygląda na to, że wartości znaczników zostały zmodyfikowane!

We wszystkich nowoczesnych przeglądarkach właściwości `id` i `action` zostały przesłonięte odwołaniami do elementów wejściowych zwyczajnie z powodu wartości `id` i `name` wybranych dla tych elementów. Pierwotne wartości właściwości odesłano już do lamusa! W przeglądarkach innych niż Internet Explorer oryginalne wartości można uzyskać za pomocą metod atrybutów modelu DOM, a w tej przeglądarce zastępowane są nawet te wartości.

Jesteśmy jednak wojownikami, którzy się nie poddadzą. Pomimo wszelkich starań twórców przeglądarek mających na celu niedopuszczenie programistów do wartości dysponujemy odpowiednią sztuczką. Możliwe jest uzyskanie dostępu do oryginalnego węzła modelu DOM, który reprezentuje sam atrybut elementu. Modyfikacje wprowadzane przez przeglądarkę nie dotyczą tego węzła. Aby uzyskać wartość z węzła atrybutu modelu DOM (np. dla atrybutu `action`), należy użyć następującego kodu:

```
var actionValue = element.getAttibuteNode("action").nodeValue;
```

W ramach ćwiczenia sprawdź, czy możliwe jest użycie tego rozwiązania do rozszerzenia metody `attr()` zawartej w kodzie z listingu 12.3, aby ustalić, kiedy atrybut węzła elementu formularza został zastąpiony przez odwołanie do elementu, a następnie określ, czy możliwe jest skorzystanie z opcji uzyskania wartości z węzła modelu DOM, gdy została ona zastąpiona.

> **UWAGA** Jeśli ciekawią Cię problemy wynikające z takich rozszerzeń elementów, polecamy sprawdzenie narzędzia DOMLint Juriya Zaytseva pod adresem *http://kangax.github.com/domlint/*, które umożliwia analizowanie strony pod kątem potencjalnych problemów. Ponadto warto zapoznać się z omówieniem Garretta Smitha w artykule *Unsafe Names for HTML Form Controls*, dostępnym pod adresem *http://jibbering.com/faq/names/*.

Choć ten problem nie może być traktowany jako błąd, ponieważ stanowi zamierzone działanie przeglądarek, ma charakter destrukcyjny. Niewątpliwie jest niepotrzebny, biorąc pod uwagę to, że odwołania do elementów można tak łatwo uzyskać za pomocą metod takich jak `document.getElementById()`.

Nie jest to jednak jedyny problem dotyczący sposobu obsługi atrybutów przez przeglądarki. Przyjrzyjmy się kolejnemu.

12.2.2. Normalizacja adresu URL

We wszystkich nowoczesnych przeglądarkach istnieje „funkcja” naruszająca zasadę minimalnego zaskoczenia: podczas uzyskiwania dostępu do właściwości, która odwołuje się do adresu URL (np. `href`, `src` lub `action`), wartość tego adresu jest automatycznie przekształcana z oryginalnej postaci w pełny, kanoniczny adres URL (nawiązano do tego na początku rozdziału).

Choć ostrzegaliśmy już o automatycznej normalizacji, napiszemy test, który demonstruje ten problem na listingu 12.5.

Listing 12.5. Demonstracja problemu z normalizacją adresu URL

```
<a href="listing-12.5.html" id="testSubject">Do samej siebie</a>

<script type="text/javascript">
  var link = document.getElementById('testSubject');

  var linkHref = link.getAttributeNode('href').nodeValue;

  assert(linkHref === 'listing-12.5.html',
         'Wartość węzła odsyłacza jest poprawna.');

  assert(link.href === 'listing-12.5.html',
         'Wartość właściwości odsyłacza jest poprawna.');

  assert(link.getAttribute('href') === linkHref,
         'Atrybut odsyłacza nie został zmodyfikowany.');

</script>
```

1. Pobiera oryginalną wartość węzła bezpośrednio z pierwszej ręki (informacje o węźle).
2. Testuje, czy oryginalna wartość węzła jest zgodna z określoną w znacznikach elementu. Test kończy się powodzeniem.
3. Testuje, czy właściwość `href` zawiera to, co jest oczekiwane, czyli tę samą wartość. Tak jednak nie jest! Ten test kończy się niepowodzeniem.
4. Sprawdza, czy wartość atrybutu jest zgodna z oczekiwaniami. Tak jest, dlatego test kończy się powodzeniem!

Kup ksi k Pole ksi k

W tym teście zdefiniowano znacznik kotwicy z atrybutem `href`, który odwołuje się do tej samej strony. Następnie uzyskiwane jest odwołanie do tego elementu w celu przeprowadzenia testu.

W dalszej kolejności stosowana jest sztuczka z poprzedniego punktu, czyli przejście do oryginalnych węzłów modelu DOM w celu znalezienia oryginalnej wartości znacznika ❶. Ta wartość jest sprawdzana ❷ przed przyjęciem „w ciemno”, że sztuczka zadziałała.

Testowana jest następnie właściwość w celu sprawdzenia, czy jest zgodna ❸. Test nie powiedzie się w żadnej przeglądarce, ponieważ wartość została znormalizowana do postaci pełnego adresu URL.

Na końcu wykonywany jest test w celu ustalenia, czy została zmodyfikowana wartość atrybutu ❹. Test kończy się pomyślnie we wszystkich przeglądarkach z wyjątkiem starszych wersji aplikacji Internet Explorer.

Te testy nie tylko prezentują charakter problemu, ale też zapewniają rozwiązanie. Możliwe jest użycie sztuczki z węzłem modelu DOM, aby uzyskać takie atrybuty, gdy wymagana jest pewność, że zostanie pobrana niezmodyfikowana wartość.

W przypadku wersji przeglądarki Internet Explorer starszych niż 8. innym rozwiązaniem jest niestandardowe rozszerzenie metody `getAttribute()`. Przekazanie magicznej liczby 2 jako drugiego parametru spowoduje, że wynikiem będzie wartość niepoddana normalizacji:

```
var original = link.getAttribute('href',2);
```

W nowoczesnych przeglądarkach można zastosować dowolne z tych rozwiązań. Sztuczka z węzłem modelu DOM zadziała w każdej przeglądarce. Wszystkie przeglądarki, z wyjątkiem aplikacji Internet Explorer, zignorują drugi parametr przekazany metodzie `getAttribute()`. Po wykonaniu takiej operacji starsze wersje przeglądarki Opera ulegną zawieszeniu bez żadnego oczywistego powodu, dlatego należy unikać tego rozwiązania, jeśli w obsługiwanym zestawie znajdują się takie wersje przeglądarki Opera.

Niewielkie jest prawdopodobieństwo tego, że problem z normalizacją adresu URL wystąpi dla utworzonego kodu, chyba że bezwzględnie wymagane jest uzyskanie przez niego wartości bez normalizacji.

Przeanalizujmy teraz problem, który może mieć znacznie poważniejsze konsekwencje.

12.2.3. Atrybut style

Atrybut `style` to ważny atrybut elementu, w przypadku którego ustawianie i uzyskiwanie wartości stanowi wyjątkowe wyzwanie. Elementy modelu DOM języka HTML oferują właściwość `style`, której można użyć w celu uzyskania informacji o stylu elementu (na przykład `element.style.color`). Jeśli jednak wymagane będzie uzyskanie oryginalnego łańcucha atrybutu `style` określonego w elemencie, okaże się to znacznie trudniejsze. Dla przykładu rozważmy następujące znaczniki:

```
<div style='color:red;'></div>
```

Co będzie, gdy pożądane jest uzyskanie oryginalnego łańcucha `color:red;`?

Właściwość `style` w ogóle nie okaże się pomocna, ponieważ jest ustawiona na obiekt, który zawiera wyniki oryginalnego łańcucha poddane analizie. Choć metoda `getAttribute("style")` zadziała w większości przeglądarek, nie będzie tak w przypadku przeglądarki Internet Explorer. Przechowuje ona w obiekcie `style` właściwość o nazwie `cssText`, której można użyć do uzyskania oryginalnego łańcucha stylu (na przykład `element.style.cssText`).

Choć bezpośrednie uzyskiwanie oryginalnej wartości atrybutu `style` może być stosunkowo rzadką operacją (w przeciwieństwie do uzyskiwania dostępu do wynikowego obiektu `style`), pojawia się inny problem z przeglądarką, który prawdopodobnie będzie mieć wpływ na dowolną stronę tworzącą elementy modelu DOM podczas działania.

12.2.4. Atrybut type

Inna pułapka występująca w wersjach 8. i starszych przeglądarki Internet Explorer dotyczy atrybutu `type` elementów `<input>`. W tym przypadku nie istnieje żadne sensowne rozwiązanie. Po wstawieniu elementu `<input>` do dokumentu jego atrybut `type` nie może już być modyfikowany. Okazuje się, że przeglądarka Internet Explorer zgłasza wyjątek po podjęciu próby zmiany tego atrybutu.

Dla przykładu przeanalizujmy kod z listingu 12.6, w którym po fakcie podejmowana jest próba zmiany typu elementu wejściowego.

Listing 12.6. Zmienianie typu elementu wejściowego po jego wstawieniu

```
<form id="testForm" action="/"></form>

<script type="text/javascript">
  window.onload = function(){

    var input = document.createElement('input');

    input.type = 'text';
    assert(input.type == 'text',
           'Typ elementu input to text.');
    document.getElementById('testForm')
            .appendChild(input);

    input.type = 'hidden';
    assert(input.type == 'hidden',
           'Typ elementu input został zmieniony na hidden.');
  };
</script>
```

❶ Tworzy nowy element, zezwalając na domyślny atrybut typu.

❷ Ustawia właściwość `type` i sprawdza ją.

❸ Wstawia nowy element `input` do modelu DOM.

❹ Zmienia typ po wstawieniu.

W tym teście tworzony jest nowy element `<input>` ❶, określany dla niego typ `text`, potwierdzane powodzenie przypisania ❷ i wstawiany nowy element do modelu DOM ❸. *Po* wstawieniu typ jest zmieniany na `hidden`, po czym sprawdzane jest, czy ta operacja faktycznie miała miejsce ❹.

Kup ksi k Pole ksi k

We wszystkich nowoczesnych przeglądarkach, z wyjątkiem programu Internet Explorer, testy bez problemu kończą się powodzeniem. Jednakże w wersjach 8. i starszych przeglądarki Internet Explorer przy próbie przypisania zgłaszany jest wyjątek, po czym drugi test nigdy nie jest wykonywany.

Choć nie ma prostego rozwiązania, możliwe jest skorzystanie z dwóch następujących środków tymczasowych:

- Zamiast próbować zmieniać atrybut `type`, utwórz nowy element `<input>`, skopiuj wszystkie właściwości i atrybuty, a następnie zastąp oryginalny element nowo utworzonym. To rozwiązanie wydaje się dość proste, ale związane są z nim problemy. Po pierwsze, niemożliwe jest ustalenie, czy element zawierał jakiekolwiek procedury obsługi zdarzeń zdefiniowane w nim przy użyciu metod drugiego poziomu modelu DOM, jeśli procedury nie były przez nas śledzone. Po drugie, wszelkie odwołania do oryginalnego elementu stają się nieważne.
- W dowolnym interfejsie API tworzonym w celu uwzględniania zmian we właściwościach lub atrybutach po prostu odrzuć wszelkie próby zmiany wartości atrybutu `type`.

Żadne z powyższych rozwiązań nie jest całkowicie satysfakcjonujące.

W bibliotece jQuery stosowane jest drugie rozwiązanie, w przypadku którego zgłaszany jest informacyjny wyjątek dla próby wprowadzenia zmiany w atrybucie `type`, jeśli element został już wstawiony do dokumentu. Choć oczywiście jest to „rozwiązanie” kompromisowe, przynajmniej interfejs użytkownika zachowuje spójność we wszystkich platformach. Na szczęście ten problem został rozwiązany w wersji 9. przeglądarki Internet Explorer.

Przyjrzyjmy się jeszcze jednemu utrudnieniu, którym nękają nas przeglądarki. Ponownie ma to związek z elementami formularza.

12.2.5. Problem z indeksem tabulacji

Określanie indeksu tabulacji elementu to kolejny dziwny problem występujący w przeglądarkach. W jego przypadku nie ma zbyt dużej zgodności w kwestii tego, jak to *powinno* działać. Choć całkowicie możliwe jest uzyskanie indeksu tabulacji elementu za pomocą właściwości `tabIndex` lub atrybutu `"tabindex"` dla elementów, w których właściwość lub atrybut zostały jawnie zdefiniowane, przeglądarka zwraca wartość `0` dla właściwości `tabIndex` oraz wartość `null` dla atrybutu `"tabindex"` elementów bez jawnie podanej wartości. Oczywiście oznacza to, że nie ma możliwości stwierdzenia, jaki indeks tabulacji został przypisany do elementów, dla których nie ustawiono jawnie wartości indeksu tabulacji.

Jest to złożona kwestia o szczególnym znaczeniu w odniesieniu do użyteczności i dostępności.

Ostatni problem dotyczący atrybutów, którym się zajmiemy, w rzeczywistości wcale nim nie jest.

Kup ksi k Pole ksi k

12.2.6. Nazwy węzłów

Choć ten problem nie jest bezpośrednio powiązany z atrybutami jako takimi, kilka rozwiązań zastosowanych w tym podrozdziale bazowało na znajdowaniu węzłów. Okazuje się, że określenie nazwy węzła może być trochę kłopotliwe.

Dokładniej rzecz biorąc, wielkość liter w nazwie węzła zmienia się w zależności od tego, jaki typ dokumentu jest rozpatrywany. Jeśli jest to normalny dokument HTML, właściwość `nodeName` zwróci nazwę elementu zawierającą wyłącznie duże litery (na przykład `HTML` lub `BODY`). Jeśli jednak jest to dokument XML lub XHTML, właściwość `nodeName` zwróci nazwę podaną przez użytkownika. Oznacza to, że nazwa może być złożona z małych lub dużych liter albo z ich kombinacji.

W przypadku tego mankamentu wygodnym rozwiązaniem jest normalizowanie nazwy przed dokonaniem jakiegokolwiek porównania (zwykle w celu uzyskania małych liter). Załóżmy, że ma zostać wykonana operacja wyłącznie dla elementów `<div>` i `<ul>`. Ponieważ nie wiadomo, czy uzyskiwane nazwy węzłów będą mieć postać `div`, `DIV` czy nawet `dIv`, wskazana będzie normalizacja nazw w sposób zaprezentowany w następującym kodzie:

```
var all = document.getElementsByTagName("*")[0];

for (var i = 0; i < all.length; i++) {
  var nodeName = all[i].nodeName.toLowerCase();
  if (nodeName === "div" || nodeName === "ul") {
    all[i].className = "found";
  }
}
```

Gdy dokładnie wiadomo, w jakiego typu dokumencie będzie wykonywany napisany kod, niekoniecznie trzeba przejmować się wielkością liter. Jeśli jednak tworzony jest kod wielokrotnego użycia, który powinien działać w dowolnym środowisku, najlepszym rozwiązaniem będzie rozwaga i przeprowadzenie normalizacji.

W tym punkcie przedstawiono problemy dotyczące atrybutów i właściwości elementów, a nawet przeanalizowano drobny problem z właściwością `style`. Jest to jednak zaledwie niewielka część tego, co przeglądarki mają w zanadrzu, jeśli chodzi o style. W następnym podrozdziale przyjrzymy się przykrym kwestiom związanym z radzeniem sobie z problemami w przeglądarkach, dotyczącym arkuszy stylów CSS.

12.3. Problemy związane z atrybutami stylów

Podobnie jak w odniesieniu do ogólnych atrybutów, uzyskiwanie i ustawianie atrybutów stylów może przysporzyć wielu kłopotów. Tak jak w przypadku atrybutów i właściwości zaprezentowanych w poprzednim podrozdziale, i tym razem dostępne są dwa sposoby obsługi wartości atrybutu `style`: wartość atrybutu oraz tworzona przy jej użyciu właściwość elementu.

Najczęściej używana jest właściwość `style` elementu, która nie jest łańcuchem, lecz obiektem przechowującym właściwości odpowiadające wartościom stylu określonym w znacznikach elementu. Oprócz tego dowiemy się, że istnieje interfejs API

Kup ksi k Pole ksi k

służący do uzyskiwania dostępu do informacji o stylu obliczanym elementu. Termin „styl obliczany" oznacza rzeczywiste style, które będą stosowane do elementu po przeanalizowaniu wszystkich dziedziczonych i użytych informacji dotyczących stylów.

W tym podrozdziale zostaną zaprezentowane rzeczy, o których należy wiedzieć przy zajmowaniu się stylami w przeglądarkach. Sprawdźmy najpierw, gdzie są rejestrowane informacje o stylach.

12.3.1. Gdzie są moje style?

Informacje o stylach umieszczone we właściwości `style` elementu modelu DOM są początkowo ustawiane na podstawie wartości określonej dla atrybutu `style` w znacznikach elementu. Na przykład kod `style="color:red;"` spowoduje, że informacje o stylach zostaną umieszczone w obiekcie style. Podczas działania strony skrypt może ustawić lub zmodyfikować wartości w tym obiekcie. Te zmiany będą aktywnie wpływać na wyświetlanie elementu.

Wielu twórców skryptów rozczarowanych jest po stwierdzeniu, że w obiekcie `style` elementu nie są dostępne żadne wartości z elementów `<style>` na stronie lub zewnętrznych arkuszy stylów. Nie pozostaniemy jednak rozczarowani zbyt długo. Wkrótce poznamy sposób uzyskiwania takich informacji.

Na razie jednak dowiedzmy się, jak właściwość `style` uzyskuje swoje wartości. Przeanalizuj poniższy kod (listing 12.7).

Listing 12.7. Analizowanie właściwości style

```
<style>
  div { font-size: 1.8em; border: 0 solid gold; }
</style>

<div style="color:#000;" title="Moc wojownika ninja!">
 忍者パワー
</div>

<script>
  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];

    assert(div.style.color == 'rgb(0, 0, 0)' ||
           div.style.color == '#000',
           'Zarejestrowano właściwość color.');

    assert(div.style.fontSize == '1.8em',
           'Zarejestrowano właściwość fontSize.');

    assert(div.style.boderWidth == '0',
           'Zarejestrowano właściwość borderWidth.');

    div.style.borderWidth = "4px";
```

1. Deklaruje arkusz stylów na stronie, który określa wielkość czcionki i informacje o ramce.
2. Ten element testowy powinien otrzymać wiele stylów z różnych miejsc, w tym z własnego atrybutu stylu oraz arkusza stylów.
3. Sprawdza, czy został zarejestrowany wstawiony styl koloru.
4. Sprawdza, czy został zarejestrowany dziedziczony styl wielkości czcionki.
5. Sprawdza, czy został zarejestrowany dziedziczony styl szerokości ramki.
6. Zastępuje styl szerokości ramki.

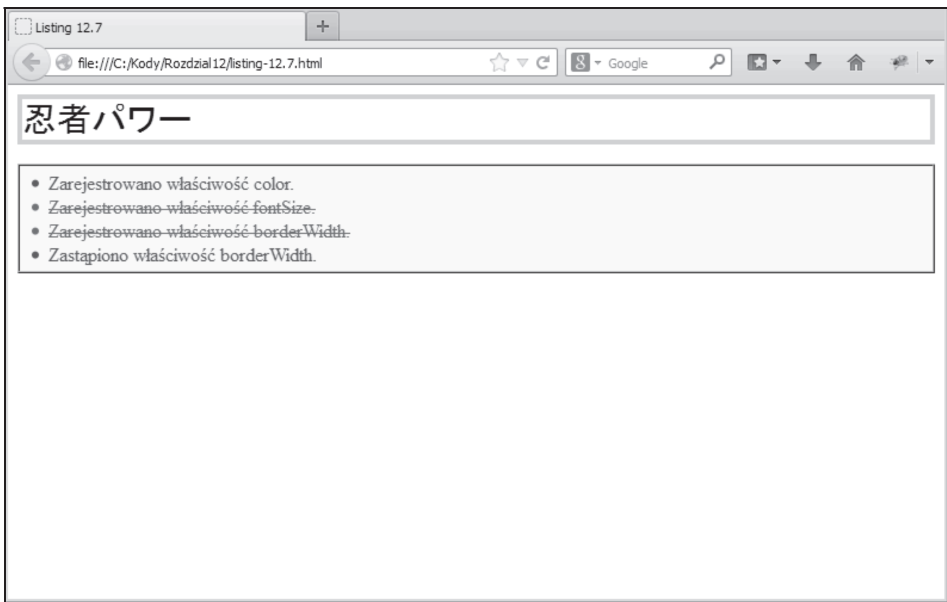
```
    assert(div.style.borderWidth == '4px',
           'Zastąpiono właściwość borderWidth.');
  };
</script>
```

❼ **Przeprowadzany jest test.**

W tym przykładzie zdefiniowano element `<style>` w celu określenia wewnętrznego arkusza stylów ❶, którego wartości będą stosowane do elementów na stronie. Arkusz stylów określa, że wszystkie elementy `<div>` zostaną wyświetlone z użyciem czcionki o wielkości, która jest 1,8 razy większa od domyślnej, a także ciągłej ramki w kolorze złotym o szerokości `0`. Oznacza to, że wszystkie elementy, dla których jest to stosowane, będą mieć ramkę. Po prostu nie będzie ona widoczna, ponieważ ma szerokość `0`.

We wstawionym atrybucie stylu tworzony jest następnie element `<div>`, który nadaje tekstowi elementu czarny kolor ❷.

Dalej rozpoczyna się testowanie. Po uzyskaniu odwołania do elementu `<div>` sprawdzane jest, czy atrybut `style` otrzymał właściwość `color` reprezentujący kolor przypisany do elementu ❸. Zauważ, że nawet pomimo tego, że we wstawianym stylu dla właściwości `color` określono wartość `#000`, w większości przeglądarek zostanie ona znormalizowana do formatu RGB po ustawieniu we właściwości `style` (z tego powodu sprawdzane są oba formaty). Jak widać na rysunku 12.3, ten test kończy się powodzeniem.



Rysunek 12.3. Testy pokazują, że rejestrowane są style wstawiane i przypisane, a style dziedziczone już nie

Kup ksi k Pole ksi k

OSTRZEŻENIE Normalizacja kolorów nie zawsze jest spójna między przeglądarkami, a nawet w obrębie jednej przeglądarki. Choć większość kolorów będzie normalizowana do formatu RGB, niektóre przeglądarki pozostawią kolory z nazwami (na przykład `black`).

Następnie naiwnie sprawdzane jest, czy w obiekcie stylu zostały zarejestrowane styl wielkości czcionki i szerokość ramki określone we wstawianym arkuszu stylów ❹ ❺. Jednakże nawet pomimo tego, że na rysunku 12.3 widać, że styl wielkości czcionki został zastosowany do elementu, test kończy się niepowodzeniem. Wynika to stąd, że obiekt stylu nie odzwierciedla żadnych informacji o stylach odziedziczonych z arkuszy stylów CSS.

W dalszej kolejności używane jest przypisanie do zmiany wartości właściwości `borderWidth` w obiekcie stylu na szerokość wynoszącą 4 piksele ❻, a następnie sprawdzane jest uwzględnienie zmiany ❼. Na rysunku 12.3 widać, że test kończy się pomyślnie, a ponadto że wcześniej niewidoczna ramka została teraz zastosowana do elementu. To przypisanie powoduje pojawienie się właściwości `borderWidth` we właściwości `style` elementu, co potwierdził test ❼.

Godne uwagi jest to, że dowolne wartości we właściwości `style` elementu będą mieć pierwszeństwo przed wszystkim, co zostało odziedziczone przez arkusz stylów (nawet jeśli reguła arkusza używa adnotacji `!important`).

Ciekawą rzeczą możliwą do zauważenia na listingu 12.7 jest to, że kod CSS określa właściwość wielkości czcionki jako `font-size`, natomiast w skrypcie użyto odwołania `fontSize`. Z czego to wynika?

12.3.2. Określanie nazw właściwości stylów

W przypadku atrybutów CSS pojawia się stosunkowo niewiele trudności związanych z uzyskiwaniem dostępu do wartości zapewnianych przez różne przeglądarki. Występują jednak różnice w sposobie określania nazw przez arkusze stylów CSS i sposobie uzyskiwania do nich dostępu w skrypcie. Ponadto istnieje kilka nazw stylów, które różnią się w poszczególnych przeglądarkach.

Atrybuty CSS, których nazwy są złożone z wielu słów, w roli separatora słów używają łącznika (np. `font-weight`, `font-size` i `background-color`). Być może pamiętasz, że nazwy właściwości w języku JavaScript *mogą* zawierać łącznik, ale zastosowanie go uniemożliwia uzyskanie dostępu do właściwości za pośrednictwem operatora kropki.

Rozważ następujący przykład:

```
var color = element.style['font-size'];
```

Powyższy kod byłby całkowicie poprawny. Z kolei następujący kod już nie:

```
var color = element.style.font-size;
```

Analizator języka JavaScript potraktowałby łącznik jako operator odejmowania. W efekcie nikt nie byłby zadowolony z wyniku. Aby nie zmuszać twórców stron do tego, by zawsze na potrzeby uzyskiwania dostępu do właściwości używali ogólnej

Kup ksi k Pole ksi k

postaci nazw stylów CSS, nazwy zawierające wiele słów, które odgrywają rolę nazw właściwości, są przekształcane w nazwy z literami o różnej wielkości. W rezultacie nazwa `font-size` przyjmuje postać `fontSize`, a nazwa `background-color` jest przekształcana w nazwę `backgroundColor`.

Albo można pamiętać o wykonaniu takiej operacji, albo napisać prosty interfejs API w celu automatycznego ustawiania lub pobierania stylów, które samoczynnie obsługują przekształcanie nazw na postać z różną wielkością liter. Prezentuje to listing 12.8.

Listing 12.8. Prosta metoda uzyskiwania dostępu do stylów

```
<div style="color:red;font-size:10px;background-color:#eee;"></div>

<script type="text/javascript">
  function style(element,name,value){                  ◄── Definiuje funkcję style.
    name = name.replace(/-([a-z])/ig,   ◄── Przekształca na postać z różną wielkością liter.
                      function(all,letter){
                        return letter.toUpperCase();
                      });

    if (typeof value !== 'undefined') {          ◄── Ustawia wartość, jeśli ją podano.
      element.style[name] = value;
    }

    return element.style[name];                         ◄── Zwraca wartość.
  }

  window.onload = function(){

    var div = document.getElementsByTagName('div')[0];

    assert(true,style(div,'color'));
    assert(true,style(div,'font-size'));
    assert(true,style(div,'background-color'));

  };
</script>
```

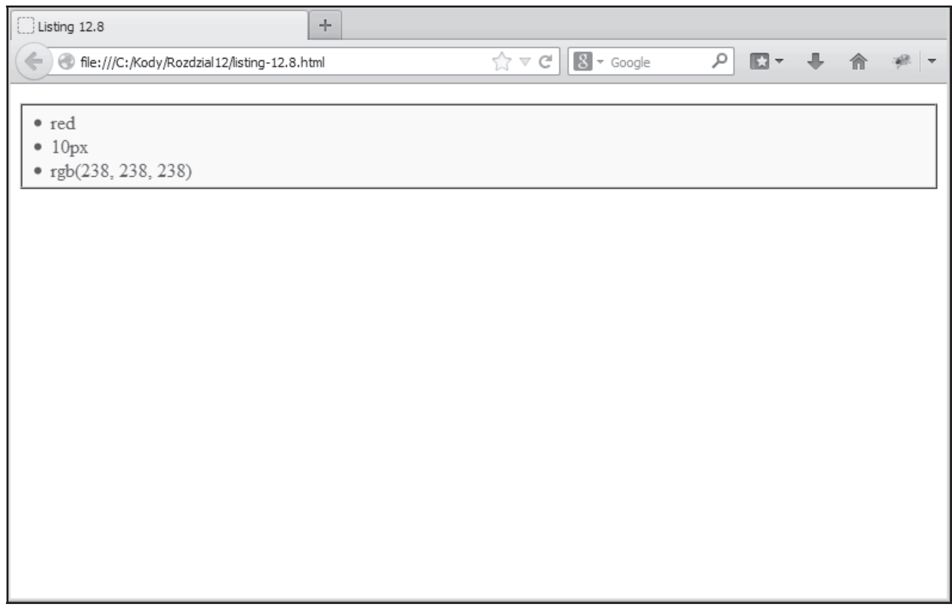
Z wyjątkiem przekształcenia parametru `name` na postać z różną wielkością liter przykładowa funkcja działa podobnie jak funkcja `attr()`, którą zaprezentowano na listingu 12.3. Z tego powodu działanie funkcji nie będzie tutaj objaśniane.

Jeśli operacja przekształcania oparta na wyrażeniu regularnym powoduje, że zaczynasz się dłużej nad nią zastanawiać, możesz przejrzeć materiał zawarty w rozdziale 7. Zauważ też, że pomimo dołączenia kilku wywołań funkcji `assert()` nie przeprowadzono w rzeczywistości żadnego testowania funkcji. Asercja posłużyła jako prosty sposób wyświetlenia danych wyjściowych na stronie (rysunek 12.4).

W ramach ćwiczenia utwórz zestaw asercji, które dokładnie przetestują tę nową funkcję.

Wcześniej wspomniano, że istnieje kilka „problematycznych" właściwości stylów, które są różnie traktowane w poszczególnych przeglądarkach. Przyjrzyjmy się jednej z nich.

Kup ksi k Pole ksi k



Rysunek 12.4. Test funkcji style() pokazuje, że może automatycznie odgadnąć nazwę właściwości dla danej nazwy CSS

12.3.3. Właściwość stylów float

W przypadku atrybutów stylu jednym z podstawowych problemów związanych z nazewnictwem jest sposób obsługi właściwości `float`. Wymaga on specjalnego traktowania, ponieważ w języku JavaScript nazwa `float` jest zastrzeżonym słowem kluczowym. Przeglądarki muszą zapewnić alternatywną nazwę.

Jak to często bywa w takich przypadkach, w przeglądarkach zgodnych ze standardami wybrano jedną drogę, a w przeglądarce Internet Explorer drugą. Jako alternatywna nazwa w niemal wszystkich przeglądarkach została użyta nazwa `cssFloat`, natomiast w programie Internet Explorer wybrano nazwę `styleFloat`. Ech.

Wykorzystując jako inspirację rozwiązanie przekształcające z listingu 12.3, sprawdź, czy możesz zmodyfikować funkcję `style()` z listingu 12.8 w celu uwzględnienia tej różnicy.

Wcześniej w rozdziale pokazano, jak można zmieniać wartości kolorów z jednego formatu w drugi, gdy są one dodawane jako właściwości stylu. Przyjrzyjmy się kolejnej takiej sytuacji.

12.3.4. Konwersja wartości pikseli

W przypadku ustawiania wartości stylu ważną kwestią do uwzględnienia jest przypisanie wartości liczbowych, które reprezentują piksele. Podczas określania wartości pikseli w nieaktualnych atrybutach, takich jak `height` znacznika `<img>`, podawano liczbę, umożliwiając przeglądarce zajęcie się jednostkami. W przypadku przypisywania wartości pikseli do właściwości stylu takie rozwiązanie może przysporzyć wielu kłopotów.

Podczas ustawiania wartości liczbowej dla właściwości stylu konieczne jest określenie jednostek, aby właściwość działała niezawodnie we wszystkich przeglądarkach. Załóżmy, że dla stylu `height` elementu ma zostać ustawiona wartość 10 pikseli. Oba poniższe wiersze kodu stanowią bezpieczny sposób zrealizowania tej operacji w różnych przeglądarkach:

```
element.style.height = "10px";
element.style.height = 10 + "px";
```

Następujący kod nie jest bezpieczny w każdej przeglądarce:

```
element.style.height = 10;
```

Możesz pomyśleć, że byłoby zwykłe dodanie niewielkiej logiki do funkcji `style()` z listingu 12.8, aby po prostu przypiąć łańcuch `px` do końca wartości liczbowej trafiającej do funkcji. Jednak nie tak szybko! Nie wszystkie wartości liczbowe reprezentują piksele! Istnieje kilka właściwości stylu pobierających wartości liczbowe, które nie reprezentują wymiaru piksela:

- `z-index`,
- `font-weight`,
- `opacity`,
- `zoom`,
- `line-height`.

W przypadku tych (oraz wszelkich innych, o których można pomyśleć) właściwości rozszerz funkcję z listingu 12.8 o automatyczną obsługę wartości, które nie reprezentują pikseli.

Ponadto przy próbie odczytu wartości piksela z atrybutu stylu powinna być używana metoda `parseFloat` do zapewnienia, że w każdej sytuacji uzyskiwana jest zamierzona wartość.

Przyjrzyjmy się zestawowi ważnych właściwości stylu, których obsługa może być kłopotliwa.

12.3.5. Określanie wysokości i szerokości

Właściwości stylu, takie jak `height` i `width`, stwarzają szczególny problem, ponieważ domyślnie, gdy nie podano wartości, ich wartość to `auto`. Oznacza to, że element dopasowuje swoją wielkość do zawartości. W efekcie właściwości stylu `height` i `width` nie można użyć do uzyskania dokładnych wartości, chyba że w łańcuchu atrybutu podano wartości jawne.

Na szczęście właściwości `offsetHeight` i `offsetWidth` przychodzą w tym przypadku z pomocą, oferując dość pewne środki uzyskiwania dostępu do faktycznej wysokości i szerokości elementu. Trzeba jednak mieć świadomość tego, że wartości przypisane do tych dwóch właściwości uwzględniają dopełnienie elementu. Taka informacja jest zwykle dokładnie tym, co jest wymagane przy próbie określenia położenia jednego elementu względem drugiego. Czasami jednak może być pożądane uzyskanie informacji o wymiarach elementu z uwzględnieniem ramek i dopełnienia albo bez nich.

Kup ksi k Pole ksi k

Kwestią, którą jednak trzeba wziąć pod uwagę, jest to, że w witrynach o wysokim poziomie interaktywności prawdopodobne jest, że przez część czasu elementy mogą znajdować się w stanie bez wyświetlania (w przypadku ustawienia dla stylu `display` wartości `none`). Gdy element nie jest wyświetlany, nie ma wymiarów. Każda próba pobrania właściwości `offsetWidth` lub `offsetHeight` dla takiego elementu spowoduje zwrócenie wartości `0`.

Aby dla takich ukrytych elementów uzyskać ich wymiary w stanie wyświetlania, można skorzystać ze sztuczki polegającej na chwilowym ujawnieniu elementu, pobraniu wartości i ponownym ukryciu go. Oczywiście powinno to zostać wykonane w taki sposób, aby nie pojawiły się żadne widoczne oznaki tego, że taka operacja jest realizowana w tle. W jaki sposób ujawnić ukryty element bez wyświetlania go?

Możemy to zrobić, korzystając ze zdobytych umiejętności wojownika! Oto niezbędne kroki:

1. Zmień wartość właściwości `display` na `block`.
2. Ustaw wartość właściwości `visibility` na `hidden`.
3. Ustaw wartość właściwości `position` na `absolute`.
4. Pobierz wartości wymiarów.
5. Przywróć zmienione właściwości.

Choć zmiana wartości właściwości `display` na `block` umożliwi pobranie rzeczywistych wartości właściwości `offsetHeight` i `offsetWidth`, sprawi, że element stanie się częścią wyświetlanej zawartości, a tym samym będzie widoczny. Aby element był niewidoczny, dla właściwości `visibility` zostanie ustawiona wartość `hidden`. Spowoduje to jednak (zawsze jest kolejne „jednak”) pozostawienie dużej luki w miejscu, w którym element został umieszczony. Z tego powodu dodatkowo dla właściwości `position` ustawiana jest wartość `absolute`, aby element nie został uwzględniony w normalnym przepływie związanym z wyświetlaniem.

Wszystko to wygląda na bardziej skomplikowane, niż jest w rzeczywistym zastosowaniu, co demonstruje listing 12.9.

Listing 12.9. Uzyskiwanie wymiarów ukrytych elementów

```
<div>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Suspendisse congue facilisis dignissim. Fusce sodales,
  odio commodo accumsan commodo, lacus odio aliquet purus,
  <img src="../images/ninja-with-pole.png" id="withPole" />
  <img src="../images/ninja-with-shuriken.png"
       id="withShuriken" style="display:none" />
  vel rhoncus elit sem quis libero. Cum sociis natoque
  penatibus et magnis dis parturient montes, nascetur
  ridiculus mus. In hac habitasse platea dictumst. Donec
  adipiscing urna ut nibh vestibulum vitae mattis leo
  rutrum. Etiam a lectus ut nunc mattis laoreet at
  placerat nulla. Aenean tincidunt lorem eu dolor commodo
  ornare.
</div>
```

Kup ksi k Pole ksi k

```
<script type="text/javascript">

  (function(){                                          ❶ Tworzy zasięg prywatny.

    var PROPERTIES = {                                  ❷ Definiuje właściwości docelowe.
      position: "absolute",
      visibility: "hidden",
      display: "block"
    };

    window.getDimensions = function(element) {          ❸ Tworzy nową funkcję.
    var previous = {};                                  ❹ Zapamiętuje ustawienia.
      for (var key in PROPERTIES) {
        previous[key] = element.style[key];
        element.style[key] = PROPERTIES[key];           ❺ Zastępuje ustawienia.
      }

      var result = {                                    ❻ Pobiera wymiary.
        width: element.offsetWidth,
        height: element.offsetHeight
      };

      for (key in PROPERTIES) {                         ❼ Odtwarza ustawienia.
        element.style[key] = previous[key];
      }
      return result;
    };

  })();

  window.onload = function() {

    setTimeout(function(){

      var withPole = document.getElementById('withPole'),
          withShuriken = document.getElementById('withShuriken');

      assert(withPole.offsetWidth == 41,                ❽ Testuje widoczny element.
             "Pobrano szerokość obrazu kija. Rzeczywista wartość: " +
             withPole.offsetWidth + ". Oczekiwano: 41");
      assert(withPole.offsetHeight == 48,
             "Pobrano wysokość obrazu kija. Rzeczywista wartość: " +
             withPole.offsetHeight + ". Oczekiwano: 48");

      assert(withShuriken.offsetWidth == 36,            ❾ Testuje ukryty element.
             "Pobrano szerokość obrazu shurikena. Rzeczywista wartość: " +
             withShuriken.offsetWidth + ". Oczekiwano: 36");
      assert(withShuriken.offsetHeight == 48,
             "Pobrano wysokość obrazu shurikena. Rzeczywista wartość: " +
             withShuriken.offsetHeight + ". Oczekiwano: 48");

      var dimensions = getDimensions(withShuriken);     ❿ Używa nowej funkcji.

      assert(dimensions.width == 36,                    ⓫ Ponownie testuje ukryty element.
             "Pobrano szerokość obrazu shurikena. Rzeczywista wartość: " +
             dimensions.width + ". Oczekiwano: 36");
```

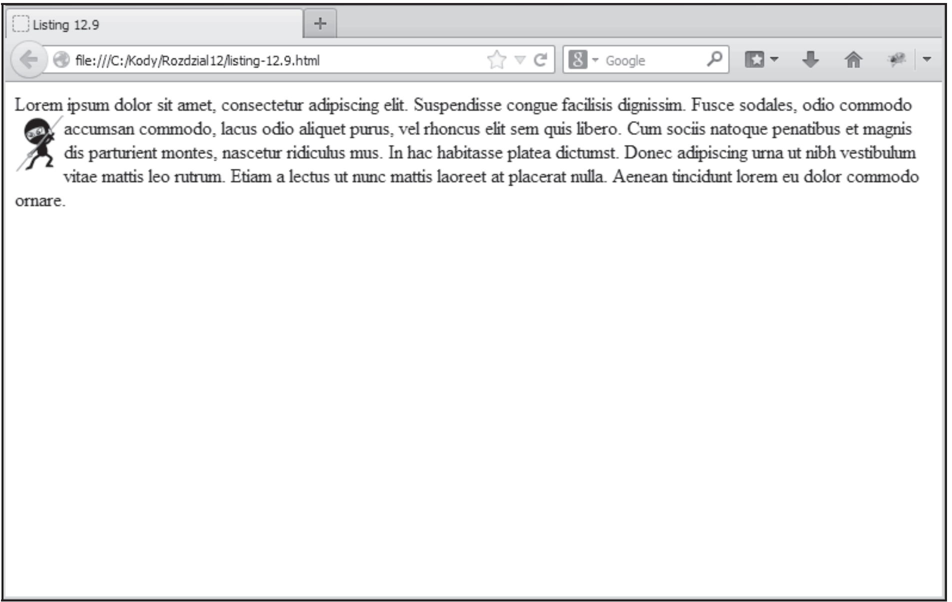
Kup ksi k Pole ksi k

```
        assert(dimensions.height == 48,
               "Pobrano wysokość obrazu shurikena. Rzeczywista wartość: " +
               dimensions.height + ". Oczekiwano: 48");
      },3000);

    }
</script>
```

Choć jest to raczej długi listing, większa jego część to kod testujący. Właściwa implementacja nowej funkcji pobierającej wymiary obejmuje swoim zasięgiem zaledwie około kilkunastu wierszy kodu.

Przeanalizujmy kod krok po kroku. Najpierw definiowanych jest kilka elementów do testowania. Element `<div>` zawiera trochę tekstu z dwoma obrazami osadzonymi w jego obrębie i wyrównywanymi do lewej strony przez style w zewnętrznym arkuszu stylów. Te elementy obrazów będą podlegać przeprowadzanym testom. Pierwszy element jest widoczny, a drugi nie.

Przed uruchomieniem jakiegokolwiek skryptu elementy są wyświetlane w sposób pokazany na rysunku 12.5. Jeśli drugi obraz nie byłby ukryty, pojawiłby się jako drugi wojownik ninja bezpośrednio po prawej stronie widocznego obrazu.



Rysunek 12.5. Zostaną użyte dwa obrazy (widoczny i ukryty) do testowania pobierania wymiarów ukrytych elementów

Dalej rozpoczyna się definicja nowej funkcji. Na potrzeby ważnych informacji zostanie użyta zmienna, dlatego zostanie ponownie zastosowana sztuczka z listingu 12.3, która polega na uwzględnieniu zmiennej lokalnej oraz definicji funkcji w funkcji bezpośredniej ❶ w celu utworzenia zasięgu lokalnego i domknięcia. Definiowana

jest zmienna lokalna, która będzie zawierać właściwości przeznaczone do modyfikowania ❷, a następnie jest wypełniana trzema właściwościami oraz ich wartościami zastępującymi.

Dalej deklarowana jest nowa funkcja pobierająca wymiary ❸ i akceptująca element, dla którego zostaną określone wymiary. W obrębie tej funkcji tworzona jest najpierw zmienna o nazwie `previous` ❹, w której będą rejestrowane poprzednie wartości właściwości stylu przeznaczone do zmodyfikowania, aby możliwe było ich późniejsze przywrócenie. W ramach pętli użytej dla właściwości zastępowania rejestrowana jest następnie każda z poprzednich wartości i zastępowana nową ❺.

Po wykonaniu tej operacji można rozpocząć określanie wymiarów elementu, który stał się częścią układu wyświetlania, lecz jest niewidoczny i ma całkowicie ustalone położenie. Wymiary są rejestrowane w zmiennej przypisanej do zmiennej lokalnej `result` ❻.

Po zdobyciu tego, czego żądano, usuwamy wszelkie ślady, odtwarzając oryginalne wartości właściwości stylu, które zostały zmodyfikowane ❼, a następnie zwracane są wyniki w postaci zmiennej zawierającej właściwości `width` i `height`.

Wszystko wspaniale, ale czy to działa? Przekonajmy się.

W procedurze obsługi ładowania przeprowadzane są testy w wywołaniu zwrotnym 3-sekundowego licznika czasu. Zapytasz, dlaczego? Taka procedura zapewnia, że test nie zostanie wykonany przed potwierdzeniem zbudowania modelu DOM. Licznik czasu umożliwia obserwowanie ekranu w trakcie działania testu w celu upewnienia się, że podczas majstrowania przy właściwościach ukrytego elementu nie wystąpiły żadne zakłócenia dotyczące wyświetlania. W końcu będzie klapa, jeśli podczas działania utworzonej funkcji dojdzie do zakłócenia w jakikolwiek sposób wyświetlania zawartości ekranu.

W wywołaniu zwrotnym licznika czasu najpierw uzyskiwane jest odwołanie do przedmiotów testów (dwa obrazy), po czym potwierdzane jest, że możliwe jest pobranie wymiarów widocznego obrazu przy użyciu właściwości przesunięcia ❽. Ten kończy się powodzeniem, o czym się można przekonać po spojrzeniu na rysunek 12.6.

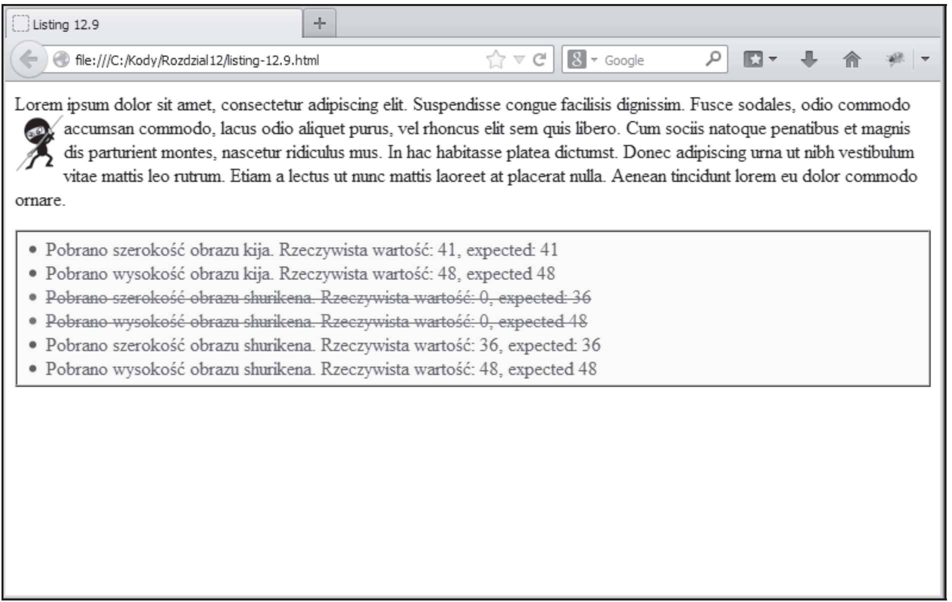
Ten sam test przeprowadzany jest dla ukrytego elementu ❾ z niewłaściwym założeniem, że właściwości przesunięcia zadziałają dla ukrytego obrazu. Niepowodzenie testu nie jest zaskoczeniem, ponieważ już wcześniej wspomniano, że tak będzie.

Dla ukrytego elementu ❿ wywoływana jest następnie nowa funkcja i ponownie jest wykonywany test ⓫. Udało się! Test kończy się powodzeniem, co potwierdza rysunek 12.6.

Jeśli w trakcie działania testu obserwowana jest zawartość strony (trzeba pamiętać o tym, że jego rozpoczęcie zostało opóźnione o trzy sekundy od momentu załadowania modelu DOM), można stwierdzić, że nie jest ona zakłócana w żaden sposób przez dokonywane w tle modyfikacje właściwości ukrytego elementu.

WSKAZÓWKA Sprawdzanie właściwości `offsetWidth` i `offsetHeight` pod kątem wartości zerowej może stanowić niebywale skuteczny sposób określania widoczności elementu.

Kup książkę Poleć książkę



Rysunek 12.6. Przez tymczasowe zmodyfikowanie właściwości stylu ukrytych elementów można z powodzeniem pobrać ich wymiary

Właściwości stylu wymiarów nie są jedynymi, które stanowią wyzwanie. Zajmijmy się niuansami związanymi z obsługą właściwości `opacity`.

12.3.6. Przenikanie nieprzezroczystości

Właściwość stylu `opacity` to kolejny szczególny przypadek, który wymaga innego traktowania w poszczególnych przeglądarkach. Choć wszystkie nowoczesne przeglądarki, w tym Internet Explorer 9, we własnym zakresie obsługują właściwość stylu `opacity`, wersje programu Internet Explorer starsze od wersji 9. używają niestandardowej notacji filtru alfa.

Z tego powodu często można się spotkać ze stylami nieprzezroczystości określonymi w arkuszu stylów w następujący sposób (lub bezpośrednio w atrybucie `style`):

```
opacity: 0.5;
filter: alpha(opacity=50);
```

Standardowy styl używa wartości z zakresu od `0.0` do `1.0`, aby określić nieprzezroczystość elementu, natomiast filtr alfa korzysta z wartości procentowej z zakresu liczb całkowitych od `0` do `100`. W obu wcześniej przedstawionych regułach wartość nieprzezroczystości jest określana na 50 procent.

Załóżmy, że w następujący sposób zdefiniowano element z obydwoma stylami:

```
<div style="opacity:0.5;filter:alpha(opacity=50);">Witaj</div>
```

Podczas próby pobrania tych wartości pojawia się problem o następujących dwóch obliczach:

- Ponieważ oprócz filtru `alpha` istnieje wiele różnych typów, takich jak transformacje, konieczne będzie radzenie sobie z wieloma typami filtrów. Po prostu nie można przyjąć, że filtr zawsze określa nieprzezroczystość.
- Nawet pomimo tego, że wersja 8. i starsze wersje przeglądarki Internet Explorer nie obsługują stylu `opacity`, określona dla niego wartość zostanie zwrócona w przypadku przywoływania właściwości `style.opacity` elementu, również wtedy, gdy zostanie całkowicie zignorowana przez przeglądarkę.

Drugi z powyższych punktów utrudnia stwierdzenie w przypadku tworzonego kodu, czy przeglądarka we własnym zakresie obsługuje styl `opacity`. Jednakże i tym razem warto skoncentrować się nad problemem, wykorzystując wszelkie zdobyte umiejętności wojownika, i zagrać na nosie przeglądarkom, które uparcie próbują krzyżować nam plany.

Jak się okazuje, przeglądarki, które obsługują styl `opacity`, zawsze będą normalizować wartość nieprzezroczystości mniejszą niż 1.0, umieszczając na początku zero. Jeśli nieprzezroczystość określono na przykład jako `opacity: .5`, przeglądarka z własną obsługą nieprzezroczystości zwróci wartość w postaci `0.5`, natomiast przeglądarki pozbawione takiej obsługi po prostu pozostawią wartość w jej oryginalnej postaci `.5`.

Oznacza to, że za pomocą symulacji funkcji (czy pamiętasz to z rozdziału 11.?) możliwe jest określenie, czy przeglądarka we własnym zakresie obsługuje nieprzezroczystość. Przeanalizuj poniższy kod (listing 12.10).

Listing 12.10. Określanie, czy przeglądarka obsługuje nieprzezroczystość

```
<img src="../images/ninja-with-nunchuks.png" style="opacity:.5;">

<script type="text/javascript">

  var div = document.createElement("div");            ← ❶ Sprawdza pod kątem obsługi.
  div.setAttribute('style','opacity:.5');
  var OPACITY_SUPPORTED = div.style.opacity === "0.5";

  assert(OPACITY_SUPPORTED,                                ← ❷ Wyświetla wyniki.
         "Nieprzezroczystość jest obsługiwana.");

</script>
```

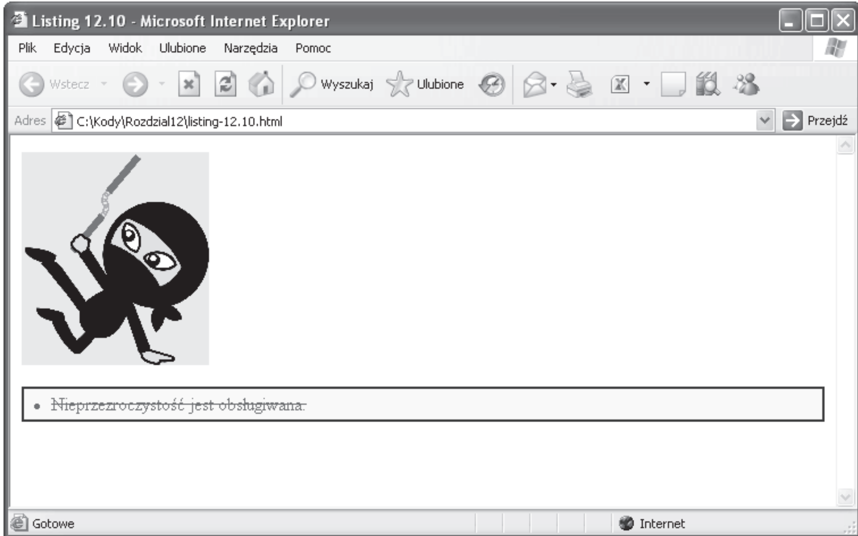
W tym przykładzie definiowany jest element obrazu ze stylem `opacity` o wartości podanej w postaci `.5`. Ten element nie zostanie użyty w kodzie. Ma on jedynie za zadanie zapewniać *nam* wizualny wskaźnik tego, czy wartość nieprzezroczystości jest uznawana przez przeglądarkę, czy nie.

Dalej rozpoczyna się kod testu, w którym tworzony jest niepowiązany element ❶. Jest on poszerzany o atrybut stylu ze stylem `opacity` o wartości `.5`. Rejestrowane jest następnie, czy nieprzezroczystość jest obsługiwana przez przeglądarkę we własnym zakresie. W tym celu wartość jest ponownie odczytywana, po czym sprawdzane jest, czy została pobrana jako oryginalna wartość (nieobsługiwana), czy jako zmodyfikowana wartość `0.5` (obsługiwana).

Kup ksi k Pole ksi k

Na końcu wykonywana jest asercja zmiennej obsługi, co powoduje powodzenie i niepowodzenie testu — odpowiednio w przypadku przeglądarek z obsługą i bez obsługi nieprzezroczystości.

Na rysunku 12.7 pokazano wynik załadowania tego testu ❷ w przeglądarkach Chrome 17 (u góry) i Internet Explorer 7 (na dole).



Rysunek 12.7. Wizualne wskazówki, a także wynik testu pokazują, że nieprzezroczystość jest obsługiwana w przeglądarce Chrome, lecz nie w wersjach przeglądarki Internet Explorer starszych niż wersja 9.

Korzystając z tej wiedzy należnej wojownikowi, sprawdź, czy możliwe jest utworzenie funkcji `getOpacity(element)` z wierszami kodu funkcji `getDimensions()` z listingu 12.9, która niezależnie od platformy zwraca wartość nieprzezroczystości dla przekazanego elementu jako wartość z przedziału od 0.0 do 1.0.

> **WSKAZÓWKA** W trakcie tworzenia tej funkcji wyrażenie regularne mogłoby okazać się pomocne w znalezieniu wartości filtru nieprzezroczystości alfa, a metoda `window.parseFloat()` będzie najlepszym towarzyszem. Dodatkowo zwróć wartość `1.0` jako wariant awaryjny, ponieważ jest to domyślne ustawienie dla wartości nieprzezroczystości.

Zajmijmy się teraz jeszcze innym zestawem problematycznych właściwości stylu, które przysparzają trochę kłopotów, gdyż ich wartości mogą przyjmować wiele równorzędnych postaci.

12.3.7. Poskromienie kolorowego koła

W rozdziale zaprezentowano już, że wartości kolorów mogą być wyrażane w różnych formatach. Powoduje to, że obsługa wartości kolorów właściwości `style` jest trochę utrudniona. W pewnym sensie jesteśmy zdani na to, jakie formaty zostaną wybrane przez twórcę strony. Co więcej, dotyczy to także transformacji stosowanych przez przeglądarki do tych formatów.

W przypadku uzyskiwania dostępu do formatów za pomocą różnych metod stylów obliczanych występuje niewielka spójność w formatach, które będą zwracane przez poszczególne przeglądarki. Z tego powodu wszelkie próby uzyskania dostępu do przydatnych składowych koloru (jego kanałów czerwieni, błękitu i zieleni, a także, jak się okaże, opcjonalnego kanału alfa) wiążą się ze sporą ilością rutynowych czynności.

Istnieje kilka formatów, w przypadku których kolory mogą być reprezentowane w nowoczesnych przeglądarkach. Zostały one zestawione w tabeli 12.3.

Na podstawie informacji zawartych w tabeli 12.3 można stwierdzić, że twórca strony dysponuje sporą elastycznością w zakresie określania informacji o kolorach. Nie powinno to stanowić zbyt dużego problemu, jeśli przeglądarki będą transformować do spójnego formatu wartości kolorów umieszczone we właściwości `style`. Tak jednak nie jest, dlatego pojawia się problem.

Napiszmy test, aby sprawdzić, jakie przeglądarki dostarczą nam zmartwień. Przeanalizuj poniższy kod (listing 12.11).

Listing 12.11. Określanie sposobu formatowania przez przeglądarkę informacji o kolorach

```
<div style="background-color:darkslateblue"> </div>
<div style="background-color:#369"> </div>
<div style="background-color:#123456"> </div>
<div style="background-color:rg6b(44,88,168)"> </div>
<div style="background-color:rgba(44,88,166,0.5)"> </div>
<div style="background-color:hsl(120,100%,25%)"> </div>
```

❶ **Tworzy elementy z kolorami.**

Tabela 12.3. **Formaty kolorów standardu CSS**

Format	Opis
keyword	Dowolne z rozpoznawanych słów kluczowych kolorów standardu HTML (red, green, maroon itp.), rozszerzone słowa kluczowe kolorów standardu SVG (bisque, chocolate, darkred itp.) lub słowo kluczowe transparent (odpowiada rgba(0,0,0,0); więcej informacji poniżej).
#rgb	Krótki format szesnastkowy RGB (*Red Green Blue*) wartości kolorów, w których każdy element jest wartością z przedziału od 0 do f.
#rrggbb	Długi format szesnastkowy RGB (*Red Green Blue*) wartości kolorów, w których każdy element jest wartością z przedziału od 00 do ff.
rgb(r,g,b)	Format RGB, w którym każda wartość dziesiętna zawiera się w przedziale od 0 do 255 lub od 0% do 100%.
rgba(r,g,b,a)	Format RGB z dołączonym kanałem alfa. Wartość alfa zawiera się w przedziale od 0.0 (przezroczystość) do 1.0 (pełna nieprzezroczystość).
hsl(h,s,l)	Format HSL (*Hue Saturation Lightness*), w którym wartości reprezentują odcień, nasycenie i jasność. Wartości odcienia należą do zakresu od 0 do 360 (kąt na kole kolorów), a wartości nasycenia i jasności zawierają się w przedziale od 0% do 100%.
hsla(h,s,l)	Format HSL z dołączonym kanałem alfa.

```
<div style="background-color:hsla(120,100%,25%,0.5)"> </div>

<script type="text/javascript">

  var divs = document.getElementsByTagName('div');   ← ❷ Gromadzi elementy.

  for (var n = 0; n < divs.length; n++) {   ← ❸ Wyświetla informacje o kolorze.
    assert(true,divs[n].style.backgroundColor);
  }

</script>
```

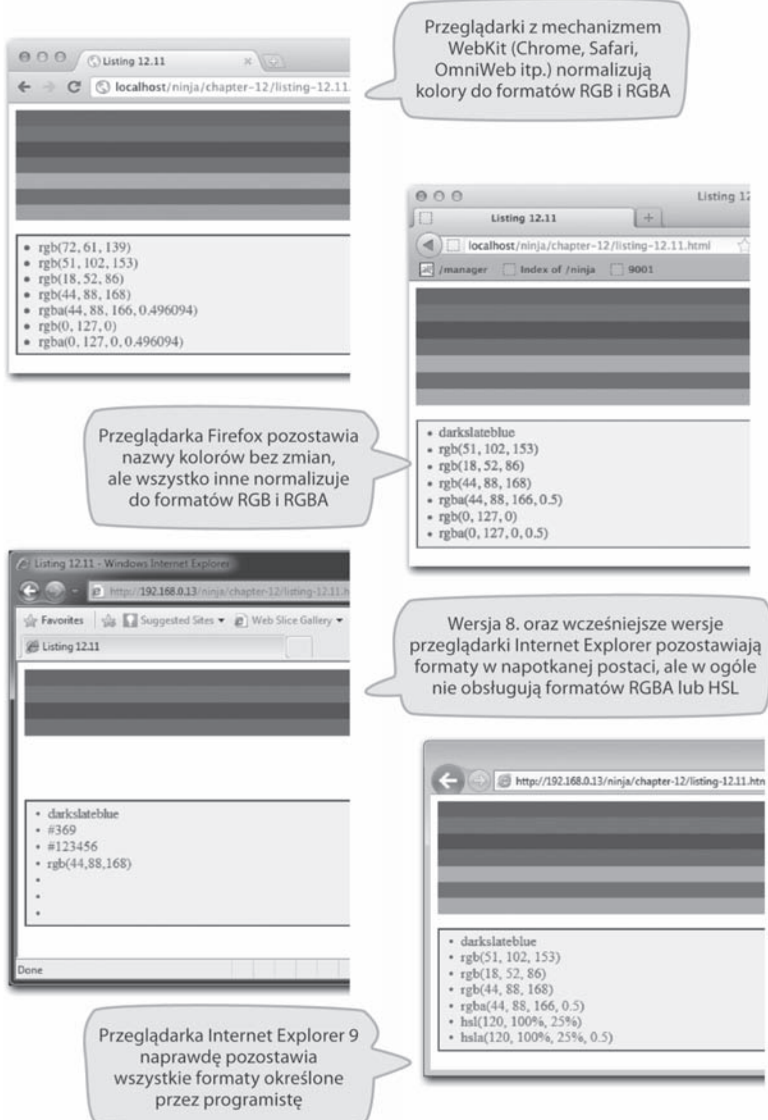
Najpierw tworzony jest zestaw elementów <div> z właściwościami style koloru tła określonymi w siedmiu różnych formatach ❶. Następnie są gromadzone odwołania do tych elementów ❷, przeprowadzana jest iteracja kolekcji oraz wyświetlana jest wartość przechowywana we właściwości style.backgroundColor ❸.

Kod umożliwia stwierdzenie, jak przeglądarka, w której test jest wykonywany, formatuje informacje o kolorach dla różnych metod ich określania. Na rysunku 12.8 widać, że zapisane formaty prezentują pełen wachlarz możliwości.

Ponieważ w poszczególnych przeglądarkach występuje tak wiele różnic dotyczących informacji o kolorach, nie będziemy tutaj zamieszczać kodu metody getColor(element,property). To zadanie pozostawimy do wykonania Tobie. Skoro dysponujesz wszystkimi niezbędnymi narzędziami, będzie to raczej czasochłonne niż trudne.

Metoda powinna akceptować element i właściwość koloru (np. color lub background-color) oraz zwracać słowo kluczowe koloru, zmienną zawierającą właściwości red, green, blue i alpha lub zmienną, która zawiera właściwości hue, lightness, saturation i alpha. Dysponując wiedzą o wyrażeniach regularnych z rozdziału 7., a także przykładami metod getDimensions() i getOpacity(), które opracowano wcześniej w rozdziale, bez obaw można zająć się tym zadaniem.

Kup ksi k Pole ksi k



Rysunek 12.8. Różne platformy przeglądarek w dość odmienny sposób radzą sobie z różnymi formatami kolorów!

WYZWANIE Jeśli naprawdę szukasz wyzwania, skonwertuj również dowolne wartości HSL do formatu RGB, używając formuły dostępnej pod adresem *http://en.wikipedia.org/wiki/HSL_and_HSV - Converting_to_RGB*.

Oczywiście obsługa kolorów nie stanowi problemu, ponieważ wcześniej się tym już zajmowaliśmy. Możliwe jest też sprawdzenie wtyczki jQuery Color z kodem napisanym przez Blaira Mitchelmore'a (*http://plugins.jquery.com/project/color*).

Kup książkę Poleć książkę

Do tej pory omówiono większość problemów, które wymagają uwzględnienia w przypadku obsługi właściwości `style` elementu. Jak jednak zaznaczono, ta właściwość nie będzie zawierać żadnych informacji o stylach, które element dziedziczy z arkuszy stylów znajdujących się w jego zasięgu. Ponieważ w wielu sytuacjach byłaby przydatna informacja o pełnym *stylu obliczanym*, który został zastosowany do elementu, sprawdźmy, czy coś takiego jest możliwe.

12.4. Uzyskiwanie stylów obliczanych

W dowolnym momencie *styl obliczany* elementu stanowi kombinację wszystkich stylów zastosowanych dla niego za pośrednictwem arkuszy stylów, atrybutu `style` elementu oraz wszelkich modyfikacji właściwości `style` przez skrypt.

Standardowy interfejs API określony przez organizację W3C, który zaimplementowano we wszystkich przeglądarkach (włącznie z przeglądarką Internet Explorer 9, lecz nie jej wcześniejszymi wersjami), to metoda `window.getComputedStyle()`. Ta metoda akceptuje element, którego style zostaną obliczone, i zwraca interfejs umożliwiający tworzenie zapytań dotyczących właściwości. Zwrócony interfejs zapewnia metodę o nazwie `getPropertyValue()`, która służy do pobierania obliczonego stylu konkretnej właściwości stylu.

W przeciwieństwie do właściwości obiektu `style` elementu metoda `getProperty↪Value()` akceptuje nazwy właściwości CSS (np. `font-size` i `background-color`), a nie wersje tych nazw o różnej wielkości liter.

Wersje przeglądarki Internet Explorer starsze niż wersja 9. udostępniają niestandardową technikę uzyskiwania dostępu do stylu obliczanego elementu. Właściwość o nazwie `currentStyle` jest dołączana do wszystkich elementów i zachowuje się bardzo podobnie jak właściwość `style`, z wyjątkiem tego, że zapewniane są informacje o aktywnym stylu obliczanym.

Dzięki temu uzyskujemy ilość informacji wystarczającą do utworzenia metody `fetchComputedStyle()`, która pobierze wartość obliczaną dowolnej właściwości stylu elementu.

Warto się zastanowić, dlaczego nie wymieniono tutaj funkcji `getComputedPro↪perty()`.

Kod z listingu 12.12 implementuje funkcję stylów obliczanych. Kod korzysta ze standardowych środków, gdy są dostępne, a w przeciwnym razie sięga po metodę niestandardową.

Listing 12.12. Pobieranie wartości stylów obliczanych

```
<style type="text/css">                          ← ❶ Definiuje arkusz stylów.
  div {
    background-color: #ffc; display: inline; font-size: 1.8em;
    border: 1px solid crimson; color: green;
  }
</style>
                                                 Tworzy przedmiot testu. ❷

<div style="color:crimson;" id="testSubject" title="Moc wojownika ninja!">
 忍者パワー
```

```
</div>

<script type="text/javascript">

  function fetchComputedStyle(element,property) {          ← ❸ Definiuje nową funkcję.

    if (window.getComputedStyle) {

      var computedStyles = window.getComputedStyle(element);   ← ❹ Uzyskuje interfejs.

      if (computedStyles) {                                  ← ❺ Pobiera wartość stylu.
        property = property.replace(/([A-Z])/g,'-$1').toLowerCase();
        return computedStyles.getPropertyValue(property);
      }
    }

else if (element.currentStyle) {                    ← ❻ Używa środków niestandardowych.
      property = property.replace(
        /-([a-z])/ig,
        function(all,letter){ return letter.toUpperCase(); });
      return element.currentStyle[property];
    }
  }

  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];

    assert(true,                                              ← ❼ Wyświetla wyniki.
           "background-color: " +
           fetchComputedStyle(div,'background-color'));
    assert(true,
           "display: " +
           fetchComputedStyle(div,'display'));
    assert(true,
           "font-size: " +
           fetchComputedStyle(div,'fontSize'));
    assert(true,
           "color: " +
           fetchComputedStyle(div,'color'));
    assert(true,
           "border-top-color: " +
           fetchComputedStyle(div,'borderTopColor'));
    assert(true,
           "border-top-width: " +
           fetchComputedStyle(div,'border-top-width'));

  };

</script>
```

Aby przetestować funkcję do utworzenia, definiowany jest element, który w swoich znacznikach określa informacje o stylu ❶, a także arkusz stylów zapewniający reguły stylów stosowane do elementu ❶. Oczekujemy, że style obliczane będą wynikiem zastosowania do elementu zarówno stylów bezpośrednich, jak i dziedziczonych.

Następnie definiowana jest funkcja akceptująca element i właściwość stylu, dla których ma zostać znaleziona wartość obliczana ❸. Aby być szczególnie przyjaznym (w końcu jesteśmy wojownikami, których jednym z zadań jest ułatwianie wszystkiego osobom korzystającym z naszego kodu), umożliwimy określenie nazw właściwości zawierających wiele słów w jednym z następujących formatów: z myślnikiem lub literami o różnej wielkości. Inaczej mówiąc, akceptowane będą nazwy `backgro↪undColor` i `background-color`. Wkrótce dowiemy się, jak to zrealizować.

Pierwszą rzeczą do zrealizowania jest sprawdzenie, czy dostępne są standardowe środki. Będzie tak w każdym przypadku z wyjątkiem starszych wersji przeglądarki Internet Explorer. Jeśli tak jest, uzyskiwany jest interfejs stylu obliczanego, który przechowywany jest w zmiennej w celu późniejszego użycia ❹. Postąpienie w ten sposób jest wymagane, ponieważ nie wiadomo, jak kosztowne może okazać się wykonanie odpowiedniego wywołania. Poza tym jest to prawdopodobnie najlepsze rozwiązanie, które umożliwi uniknięcie niepotrzebnego powtarzania wywołania.

Jeśli się to powiedzie (choć nie przychodzi nam na myśl żaden powód, dla którego miałoby być inaczej, często warto być przezornym), zostanie wywołana metoda `get↪PropertyValue()` interfejsu w celu uzyskania wartości stylu obliczanego ❺. Najpierw jednak zmieniana jest nazwa właściwości, aby dostosować ją do wersji nazwy z łącznikiem lub wersji ze znakami o różnej wielkości. Ponieważ metoda `getProper↪tyValue()` oczekuje wersji nazwy z łącznikiem, użyta zostanie metoda `replace()` obiektu `String` z prostym, ale sprytnym wyrażeniem regularnym, aby łącznik wstawić przed każdą dużą literą, po której znajdą się wyłącznie małe litery (możemy się założyć, że będzie to łatwiejsze, niż myślisz).

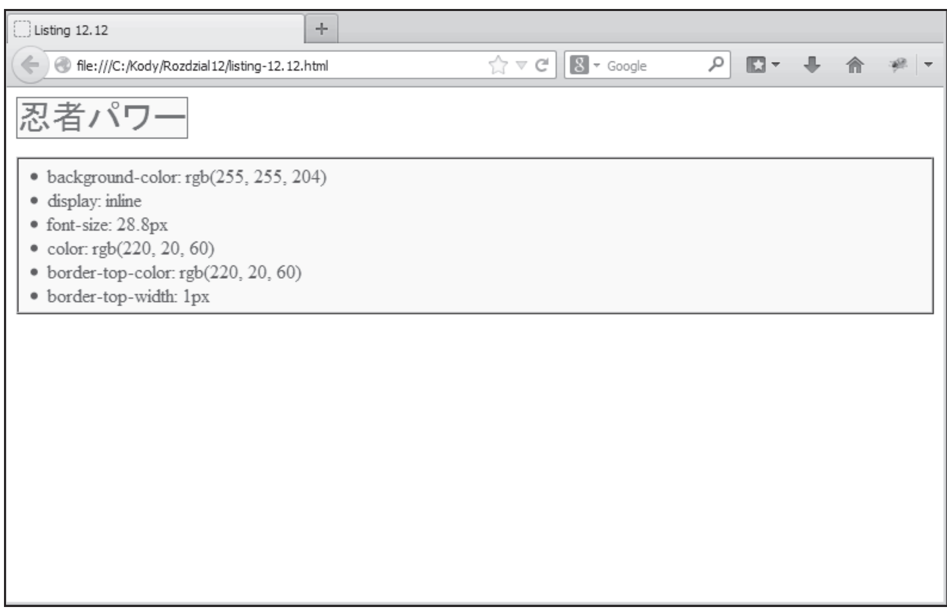
Jeśli zostanie stwierdzone, że metoda standardowa jest niedostępna, sprawdzana jest dostępność niestandardowej właściwości `currentStyle` przeglądarki Internet Explorer. Jeśli tak jest, następuje transformacja nazwy właściwości przez zastąpienie odpowiednikiem w postaci dużej litery wszystkich wystąpień małych liter poprzedzonych łącznikiem (w celu przekształcenia wszystkich nazw właściwości zawierających łącznik w nazwy z literami o różnej wielkości), po czym zwracana jest wartość tej właściwości ❻.

Jeśli coś się nie powiedzie, w każdym przypadku po prostu nie zostanie zwrócona żadna wartość.

W celu przetestowania funkcji wykonywanych jest kilka jej wywołań z przekazaniem różnych nazw stylów w odmiennych formatach, a następnie wyświetlane są wyniki ❼ (rysunek 12.9).

Zauważ, że style są pobierane niezależnie od tego, czy zostały jawnie zadeklarowane w elemencie, czy odziedziczone z arkusza stylów. Zwróć również uwagę na fakt, że właściwość `color`, którą określono zarówno w arkuszu stylów, jak i bezpośrednio w elemencie, zwraca jawną wartość. Style określone przez atrybut `style` elementu zawsze mają pierwszeństwo nad stylami dziedziczonymi, nawet gdy oznaczono je przy użyciu dyrektywy `!important`.

W przypadku zajmowania się właściwościami stylów konieczna jest świadomość jeszcze jednego zagadnienia, czyli właściwości *połączonych*. Standard CSS umożliwia użycie skróconego zapisu w przypadku połączenia właściwości (np. właściwości



Rysunek 12.9. Style obliczane obejmują wszystkie style określone w elemencie, a także style dziedziczone z arkuszy stylów

zawierające w nazwie łańcuch `border-`). Zamiast określać kolory, szerokości i style osobno dla poszczególnych czterech ramek oraz dla nich wszystkich, można zastosować następującą regułę:

```
border: 1px solid crimson;
```

Dokładnie taka reguła została użyta na listingu 12.12. Choć pozwala ona uniknąć nadmiernego wpisywania, trzeba mieć świadomość tego, że podczas pobierania właściwości wymagane jest uzyskanie poszczególnych właściwości na najniższym poziomie. Nie można pobrać właściwości `border`, ale style, takie jak `border-top-color` i `border-top-width`, już tak. Właśnie w ten sposób postąpiono w przedstawionym przykładzie.

Choć może to być trochę kłopotliwe, zwłaszcza gdy wszystkie cztery style mają takie same wartości, trzeba sobie z tym radzić.

12.5. Podsumowanie

W przypadku problemów ze zgodnością z różnymi przeglądarkami być może pobieranie i ustawianie atrybutów, właściwości i stylów modelu DOM nie jest najgorszą częścią programowania w języku JavaScript pod kątem przeglądarek, ale z pewnością ma w całości spory udział. Na szczęście dowiedzieliśmy się, że z tymi problemami można sobie poradzić w sposób zapewniający zgodność z różnymi przeglądarkami, który nie wymaga uciekania się do wykrywania przeglądarki.

Oto istotne kwestie poruszone w tym rozdziale:

Kup ksi k Pole ksi k

- *Wartości* atrybutów są ustawiane przy użyciu atrybutów umieszczonych w znacznikach elementu.
- Po pobraniu wartości atrybutów mogą reprezentować te same wartości, lecz czasem mogą być sformatowane w inny sposób niż w oryginalnych znacznikach.
- Właściwości, które reprezentują wartości atrybutów, tworzone są w elementach.
- Klucze dla tych właściwości mogą różnić się od nazwy oryginalnego atrybutu. Ponadto w poszczególnych przeglądarkach wartości mogą być formatowane inaczej niż wartość atrybutu lub oryginalne znaczniki.
- Gdy zaistnieje taka potrzeba, oryginalną wartość znaczników można pobrać przez podzielenie ich na węzły oryginalnych atrybutów w modelu DOM i uzyskanie z nich wartości.
- Korzystanie z właściwości jest zwykle efektywniejsze niż używanie metod atrybutów modelu DOM.
- Wersje przeglądarki Internet Explorer starsze niż wersja 9. nie zezwalają na modyfikację atrybutu `type` elementów `<input>`, gdy są już one częścią modelu DOM.
- Z atrybutem `style` związanych jest kilka specyficznych trudności. Ponadto atrybut nie zawiera stylu obliczanego dla elementu.
- W nowoczesnych przeglądarkach style obliczane mogą być pobierane z obiektu `window` za pomocą standardowego interfejsu API. Z kolei w wersjach 8. i starszych przeglądarki Internet Explorer umożliwia to właściwość niestandardowa.

W rozdziale omawiane były problemy wywoływane przez stosowanie różnych implementacji sposobu obsługi właściwości i atrybutów przez różne przeglądarki. Dowiedzieliśmy się, że w tej dziedzinie występuje naprawdę sporo trudności. Być może jednak w obszarze tworzenia aplikacji internetowych nie ma sfery, z którą byłoby związanych tyle problemów dotyczących wsparcia wielu przeglądarek, ile występuje w obrębie obsługi zdarzeń. Tym zagadnieniem zajmiemy się w następnym rozdziale.

Kup ksi k Pole ksi k

Skorowidz

A

adres URL, 317
analizowanie
kodu, 240, 402
bezpieczeństwo, 247
konstruktor Function, 243
liczniki czasu, 244
metoda eval(), 240
model DOM, 247
przekształcanie łańcuchów, 251
zasięg globalny, 244, 246
selektora, 421
wydajności, 31
animacje, 129, 232, 234
antyfunkcja, 287
anulowanie powiązań, 346, 350, 364
argumenty funkcji, 72
arkusze stylów, 288, 305
asercja, 44, 325
atrybut, 305
style, 318, 340
type, 319
atrybuty
modelu DOM, 307
niestandardowe, 310
stylów, 321
automatyczne zapamiętywanie, 141

B

bezpieczne analizowanie kodu, 247
biblioteka
base2, 183, 253, 271
Functional.js, 138
jQuery, 128, 148, 152, 284
jQuery U, 109
Prototype, 24, 135–138, 143, 172, 270
YUI, 272
błędy, 180
nieprzewidywalne, 291
w przeglądarkach, 281
buforowanie, 102

C

centralne przechowywanie informacji, 355
CSV, Comma-Separated Value, 136
czas
ładowania biblioteki, 255
oczekiwania, 223
czasochłonne przetwarzanie, 229
częściowe stosowanie funkcji, 138
czyszczenie pamięci, 123
czytelność kodu, 149

D

debugowanie kodu, 34, 38
definiowanie typów obiektów, 155
deklaracje
funkcji, 63
zmiennych, 67
dekompilacja funkcji, 248–251
delegowanie zdarzeń, 375, 390
diagram
czasowy, 221
relacji, 93
długotrwałe zadanie, 229
dodatek Firebug, 271
dodawanie
dziedziczenia, 166
obsługi zdarzeń, 385
właściwości, 174
domknięcia, 114, 119–154
dopasowanie
lokalne, 206
globalne, 206
dopasowywanie, 205
znaków, 198, 214
znaków Unicode, 216
dostęp do
atrybutów, 142, 307
funkcji, 165
kolekcji, 109
metod, 117
stylów, 325
właściwości, 162, 311
zmiennych, 130, 271
DSL, Domain-Specific Language, 259
dynamiczne
aktualizacje, 162
przebudowywanie kodu, 256
działanie
atrybutów niestandardowych, 310
domknięć, 120
liczników czasu, 220
selektorów, 412
dziedziczenie, 166, 183
dziedziczenie prototypów, 168
dzielenie listy argumentów, 110

E

element
<button>, 132
<div>, 128, 308
<style>, 322
elementy
animowania, 234
HTML, 172
skryptowe, 410
tablicy, 425
ukryte, 328, 330

F

faza
propagacji, 347
przechwytywania, 347
FIFO, first in, first out, 57, 222
filtr alfa, 332
filtrowanie zestawu, 423
flagi, 196
format CSV, 136
formaty kolorów, 336, 337
fragmenty modelu DOM, 399

- funkcja, 51, 84
  - addMethod(), 113
  - animateIt(), 130
  - getData(), 358
  - hasOwnProperty(), 286
  - innerFunction(), 122
  - isFunction(), 117
  - merge(), 108
  - readAttribute(), 142
  - removeEvent(), 366
  - wrap(), 143
- funkcje
  - anonimowe, 88, 94, 118
  - bezpośrednie, 144, 147, 151
  - jako metody, 96
  - jako obiekty, 56, 98
  - nieglobalne, 66
  - przeciążane, 107
  - rekurencyjne, 90, 92, 118
  - wewnętrzne, 122, 124
  - wstawiane, 96, 97
  - wywołania zwrotnego, 233
  - z automatycznym zapamiętywaniem, 100
  - z nazwą, 65, 91, 95
- funkcji
  - deklaracje, 63
  - dekompilacja, 248
  - lista parametrów, 85
  - określanie zasięgu, 66
  - serializacja, 186
  - wymuszanie kontekstu, 82
  - wywołania, 71, 74, 85
  - zasięg, 67
  - zastąpienie w dopasowaniu, 210
- funkcjonalności, 289
- funkcyjność, 54

G

- generowanie
  - modelu DOM, 398
  - strony HTML, 273
  - testu, 38
- globalna przestrzeń nazw, 180
- globalne wyrażenia regularne, 205
- gromadzenie skryptów, 402
- grupy testów, 45

H

- hermetyzacja
  - informacji, 125
  - kodu, 284

I

- identyfikator GUID, 357
- implementacja
  - modelu DOM, 418
  - magazynu obiektów, 355
  - zdarzenia gotowości, 388
  - zdarzeń, 383, 384
- importowanie
  - kodu, 253, 272
  - przestrzeni nazw, 253
- indeks
  - dopasowania, 210
  - tabulacji, 320
- informacje
  - o kolorach, 335
  - o stylach, 322
- inicjalizacja podklas, 187
- instancja
  - funkcji, 116
  - klasy, 179
  - obiektu, 164
  - superklasy, 188
- instrukcja
  - for-in, 109
  - typeof, 116
  - with, 263–276
- instrukcje rejestrowania, 34
- interfejs
  - API, 55, 290, 300, 412
  - API selektorów, 413
  - CGI, 231
- interwał, 223

J

- język
  - HTML, 310
  - JavaScript, 13, 25
  - Objective-C, 261
  - Objective-J, 260
  - Processing.js, 259
  - XML, 310
  - XPath, 416
- języki DSL, 258

K

- klasa, 183
- klonowanie elementów, 403, 405
- kod
  - CSS, 324
  - HTML, 172, 394
  - wielokrotnego użycia, 277, 303
  - XML, 396
- kody źródłowe, 16
- kolejka FIFO, 57, 222
- kolejność
  - arkuszy stylów, 288
  - inicjalizatorów, 159
- kolekcje, 103
- kolory, 335
- komparator, 60
- kompilowanie wyrażeń regularnych, 201
- kompresja kodu, 254
- konstrukcja !!, 100
- konstruktor, 77–79, 157, 190
- konstruktor Function, 243
- kontekst funkcji, 131
- kontrolowanie liczników czasu, 231
- konwersja wartości pikseli, 326
- kosztowne obliczenia, 101

L

- liczba argumentów, 112, 114
- liczniki czasu, 127, 219
- lista parametrów, 85, 105, 109
- literał funkcji, 63
- luźne sprzężenie, 371

Ł

- ładowanie
  - biblioteki, 255
  - strony, 367
- łańcuch, 211
  - JSON, 252
  - prototypów, 166–169

M

- magazyn obiektów, 355
- mankamenty przeglądarek, 390
- metajęzyki, 258
- metoda, 95
  - _super, 190
  - add(), 105
  - apply(), 80, 105, 178
  - bind(), 135
  - call(), 80, 84, 106
  - eval(), 240–242
  - exec(), 206
  - feint(), 126
  - forEach(), 170
  - getElementsByClassName(), 418
  - match(), 205

Kup ksi k   Pole ksi k

metoda
memoized(), 139, 142
querySelector(), 413
range.createContextual
↪Fragment(), 394
replace(), 208, 209
slice(), 109, 110
split(), 136
subClass(), 184
trim(), 214
unique(), 425
metody
przeciążone, 116
tablicowe, 103
zapamiętywania, 139, 141
model
automatu
obliczeniowego, 195
DOM, 102, 171, 307, 393
fragmenty, 399, 400
klonowanie elementów, 403
kod HTML, 394, 395
poziom drugi, 389
poziom zerowy, 389
przekształcanie
kodu HTML, 396
przekształcanie
łańcucha, 407, 410
punkt wstawienia, 401
usuwanie elementów, 410
wstawianie fragmentu, 400
wykonywanie skryptu, 401
IE, 389
modele obsługi zdarzeń, 389
modyfikowanie
arkuszy stylów, 172
kontekstu funkcji, 131
modelu DOM, 393
znaczników, 316

N

narzędzia do testowania, 42
narzędzie
Firebug, 34
IE Developer Tools, 34
JsUnit, 41, 43
Opera Dragonfly, 34
Packer, 255
QUnit, 41, 43
Selenium, 41
WebKit Developer T, 34
YUI Compressor, 256
YUI Test, 41, 43
nawiasy
definiujące
przechwytywanie, 209
grupujące, 209
okrągłe, 63, 145, 205, 208
nazwy
argumentów, 249
atrybutów, 308
parametrów, 147
węzłów, 321
właściwości, 309
właściwości stylów, 324
niepoprawne odwołania, 182
nieprzezroczystość, 205, 333
normalizacja
adresu URL, 317
kolorów, 324

O

obcinanie łańcucha, 212
obiekt, 162
Array, 103, 170, 177
Event, 351, 362
Object, 167
RegExp, 196
obiekty pierwszej klasy, 54
Objective-J, 260
obniżenie poziomu
funkcjonalności, 289
obsługa
błędów, 282
czasu oczekiwania, 223
gotowości dokumentu, 390
interwału, 223
przeglądarek, 26, 28, 277, 279,
Patrz także przeglądarki
kod zewnętrzny, 283
kwestie programistyczne,
279
liczba założeń, 301
nieprzewidywalne błędy, 291
obniżenie
funkcjonalności, 289
odbiorca docelowy, 280
regresje, 290
strategie
implementowania, 292
zgodność wstecz, 289
znaczniki, 283
selektorów, 411
stopniowana przeglądarek, 27
zdarzeń, 57, 59, 133, 134
anulowanie powiązań, 364
test dymu, 367
wiązanie procedur, 359
zarządzanie procedurami,
355, 358
zdarzeń przeglądarki, 132
odwołania, 159
bez przedrostka, 267
do obiektu, 160
pośrednie, 92
właściwości, 163
wsteczne, 200, 207
złożone, 270
ograniczenia dotyczące nazw, 309
opakowanie
biblioteki, 152
funkcji, 142
kodu html, 397
opakowująca funkcja
anonimowa, 144
operator
instanceof, 165, 181
new, 126, 157, 179
operatory zachłanne, 199
opóźnianie wykonywania kodu, 49
opóźnienie licznika czasu, 225
organizacja W3C, 377

P

pakiet testów, 44
parametr
$, 148
arguments, 73
this, 73, 158
parametry jawne, 85
pętla zdarzeń przeglądarki, 56–58
pętle, 150
pobieranie tekstu, 409
podklasy, 177, 185, 187
podwójne przypisanie, 152
podział łańcucha, 137
poprawki błędów, 281, 283
porównanie wydajności, 311
predefiniowane
klasy znaków, 199
właściwości, 158
procedury obsługi zdarzeń, 358,
364, 390
Processing.js, 259
programowanie aspektowe, 257
propagacja, 375
zdarzenia change, 381
zdarzenia submit, 378

prototypy, 135, 155, 191
prototypy modelu DOM, 171
przechodzenie list argumentów, 107
przechowywanie
  danych, 358
  funkcji, 99
przechwytywanie
  dopasowania, 210
  wartości, 204
przeciążanie funkcji, 105–118
przeglądarki, 26, 28, 278
  aktywowanie zdarzenia, 299
  antyfunkcja, 287
  awarie, 300
  bezpieczne poprawki, 292
  błędy, 281
  brakujące funkcje, 289
  implementacja modelu DOM, 315
  interfejsy API, 300
  liczba założeń, 301
  nieprzewidywalne błędy, 291
  obsługa błędów, 281, 283
  obsługa zdarzeń, 377
  powiązania procedur, 299
  regresje, 290
  symulacja funkcji, 295
  właściwości arkuszy stylów, 300
  wydajność interfejsu API, 301
  wykrywanie obiektu, 294
  wymuszanie reguł, 315
  żądania Ajax, 301
przekształcanie
  funkcji, 117
  kodu HTML, 396
  łańcucha, 210, 410
  łańcucha JSON, 251
przesłanianie
  działania funkcji, 139
  metod, 143
przetwarzanie
  łańcucha, 396
  obliczeniowe, 228
  wstępne, 252
przypisania, 266
przywoływanie przechwytywań, 207
pułapki, 173, 180
punkt
  wstawienia, 401
  wstrzymania, 36, 128

R

regresje, 290
rejestrowanie, 34
rekurencja, 90–93, 424
rozszerzanie
  liczby, 175
  nazwy, 315
  obiektu, 173
rozwijanie funkcji, currying, 136

S

scalanie, 424
scalanie argumentów, 138
selektory CSS, 411, 416
  analizowanie, 421
  mechanizm wstępujący, 425
  mechanizm zstępujący, 420
  projektowanie mechanizmu, 419
  wyrażenie regularne, 421
selektory CSS3, 418
serializacja funkcji, 186
skrypt zorientowany aspektowo, 257
słowo kluczowe
  function, 63
  new, 156
sortowanie, 54, 60, 61
specyfikacja HTML5, 172
sprawdzanie procedur obsługi, 363
standard ECMAScript 5, 177
stosowanie
  częściowe funkcji, 136
  poprawek, 293
  szablonów, 274, 275
struktura obiektu, 162
styl obliczany, 338, 341
superklasa, 187
symulowanie
  funkcji, 295, 298
  tablic, 104
  zdarzenia, 369
szablony, 273, 275

Ś

środowiska testowania, 40, 43

T

tablice, 103
techniki przeciążania funkcji, 111
tekst
  pobieranie, 409
  ustawianie, 408
test, 65
  dymu, 366, 368
  jednostkowy, 40, 43
  wydajności, 269, 312
testowanie, 30, 38, 42
  asynchroniczne, 47, 235
  dziedziczonych właściwości, 286
  funkcji, 66, 114, 272
  funkcji style(), 326
  interfejsu API, 349
  konstruktywne, 39
  wartości zwróconych, 242
treść tekstowa, 407
tworzenie
  aliasu złożonych odwołań, 270
  domknięć, 131
  instancji, 156, 178
  liczników czasu, 223
  testów, 40
typ
  funkcji, 116
  obiektu, 164

U

ukrywanie kodu, 254
unikanie właściwości, 285
ustawianie
  liczników czasu, 221
  tekstu, 408
  właściwości outerHTML, 406
usuwanie
  elementów, 405, 410
  licznika czasu, 220
używanie
  domknięć, 125
  licznika czasu, 229
  podklas, 178, 185

W

wartości
  atrybutów, 342
  stylów obliczanych, 338, 340
  znaczników, 342
wartość
  boolowska, 100
  undefined, 160
wątki, 219

Kup ksi k Pole ksi k

węzły, 321, 342, 393, 399
wiązanie
procedur, 347
zdarzeń, 346, 350, 359, 362
wielkość czcionki, 324
właściwości, 159, 342
dziedziczone, 286
funkcji, 103, 118
instancji, 158
modelu DOM, 307
osadzane, 285
połączone, 340
stylów, 324, 327
właściwość
callee, 97, 181
color, 323
constructor, 164
filter, 205
float, 326
height, 327
innerText, 407
length, 111, 177
offsetWidth, 331
opacity, 332
prototype, 190
src, 306
tabIndex, 320
textContent, 407
wstępne przetwarzanie łańcucha, 396
wstępujący mechanizm selektorów, 425
wybór przeglądarek, 278
wydajność, 31, 102, 268, 311, 393
wydajność licznika czasu, 226
wykonywanie skryptu, 401
wykrywanie
obiektu, 294
propagacji zdarzeń, 377
wymiary ukrytych elementów, 328
wymuszanie reguł, 315
wymuszenie kontekstu funkcji, 82
wyrażenia, 145
wyrażenia regularne, 193–217
alternatywa, 200
dopasowywanie, 205, 214
flagi, 196
globalne, 205
grupowanie, 199, 208
kompilowanie, 201
obcinanie łańcucha, 212
odwołania wsteczne, 200, 207
opcje powtarzania, 198
operatory, 197
przechwytywanie, 204, 208
przekształcanie łańcucha, 210
wyrażenia XPath, 416
wywołania zwrotne, 59, 83, 95, 127
wywołanie funkcji, 71, 76, 85
jako funkcji, 73
jako konstruktora, 77
jako metody, 74
metoda apply(), 80
metoda call(), 80
wyzwalanie zdarzeń, 369, 390
wyzwalanie zdarzeń niestandardowych, 372
wzorce, *Patrz* wyrażenia regularne

Z

zachłanne identyfikatory, 287
założenia, 301
zapamiętywanie, memoization, 100
elementów modelu dom, 102
funkcji, 141
obliczonych wartości, 101
odwołań, 150
zarządzanie
procedurami, 355
zdarzeniami, 345–391
zasięg
autonomiczny, 146
deklaracji, 85
funkcji, 66, 122
globalny, 181
instrukcji with, 264, 269
ograniczony, 149
tymczasowy, 146
zmiennych, 69
zastosowanie
domknięć, 126
interfejsu API selektorów, 414
zdarzenia, 58, 345–391
zdarzenia niestandardowe, 371–375, 390
zdarzenie
change, 381
focusin, 383
focusout, 383
gotowości dokumentu, 387
mouseenter, 384
mouseleave, 384
propagacji, 369
submit, 378, 380
zgodność wstecz, 289
zmiana znaczenia znaków, 198
zmiany
w interfejsie API, 290
w prototypie, 161
zmienne prywatne, 125, 146
znaczniki, 283
znaczniki skryptu, 257
znajdowanie elementów, 416, 422
znak
$, 128, 198
/, 196
łącznika, 198
nowego wiersza, 214
znaki
\\, 196
o zmienionym znaczeniu, 216
Unicode, 215
zwolnienie wątku, 222
zwracanie przechwytywań, 209

Ż

żądania Ajax, 301, 372

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

JAVASCRIPT to język programowania, który wymaga od programisty szerokiej wiedzy i dokładności. Chwila nieuwagi może spowodować poważne problemy, trudne do wykrycia. Jak sobie radzić w tym wymagającym środowisku? Jak zwinnie poruszać się pomiędzy zastawionymi pułapkami?

Odpowiedzi na te i wiele innych pytań znajdziesz w tej książce. Dzięki niej będziesz zwinnie jak ninja przemykał pomiędzy niuansami języka JavaScript. W trakcie lektury poznasz dogłębnie najlepszą broń przeciw błędom — debuger oraz testy automatyczne. W kolejnych rozdziałach nauczysz się korzystać z potencjału funkcji oraz domknięć. W tej doskonałej książce znajdziesz również szerokie omówienie wyrażeń regularnych — tematu, który spędza programistom sen z oczu. Ponadto szczegółowo poznasz zasady programowania obiektowego w JavaScripcie, modyfikowania drzewa DOM, wsparcia dla różnych przeglądarek oraz obsługi zdarzeń. Książka ta zawiera praktyczne porady, które sprawdzą się w codziennej pracy z językiem JavaScript. Jest to obowiązkowa pozycja dla każdego programisty!

DZIĘKI TEJ KSIĄŻCE:

- **nauczysz się pisać testy automatyczne**
- **wykorzystasz wyrażenia regularne w JavaScripcie**
- **zmodyfikujesz drzewo DOM**
- **opanujesz niuanse języka JavaScript**

PISZ KOD JAVASCRIPT JAK PRAWDZIWY NINJA!

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

Informatyka w najlepszym wydaniu