

JavaScript

Praktyczny kurs

Zostań specjalistą
w tworzeniu interaktywnych
stron internetowych!

- **Jak zapewnić** interaktywne zachowanie stron WWW?
- **Jak korzystać** ze zmiennych, operatorów, instrukcji oraz pętli?
- **Jak stworzyć** atrakcyjną, bezawaryijną witrynę?

Marcin Lis



Helion

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiekolwiek 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 właścicielami.

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

Redakcja: Krzysztof Zemanek

Projekt okładki: Maciej Pasek

Fotografia na okładce została wykorzystana za zgodą iStockPhoto Inc.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 032 231 22 19, 032 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie?jscpk_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

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

ISBN: 978-83-246-5428-4

Copyright © Helion 2009

Printed in Poland.

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

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

Spis treści

Wstęp	7
Rozdział 1. Pierwsze kroki	9
Lekcja 1. Pierwszy skrypt	9
Pierwszy skrypt	9
HTML czy XHTML?	11
Umieszczanie skryptów w kodzie HTML i XHTML	11
Standardy JavaScript	17
JavaScript i Java	17
Rozdział 2. Instrukcje języka	19
Lekcja 2. Instrukcje, zmienne i typy danych	19
Struktura kodów HTML i XHTML w tym rozdziale	19
Czym są instrukcje?	21
Co to jest zmienna?	22
Typy danych w JavaScriptie	23
Znaczniki HTML	27
Komentarze	28
Uwagi dotyczące struktury leksykalnej	30
Instrukcja document.write	32
Lekcja 3. Operacje i operatory	33
Wykonywanie operacji	33
O wyświetlaniu danych raz jeszcze	35
Operatory arytmetyczne	36
Operatory inkrementacji i dekrementacji	38
Operatory porównywania (relacyjne)	41
Operatory logiczne	41
Operatory bitowe	43
Operatory przypisania	46
Operator warunkowy	48
Operator typeof	48
Pozostałe operatory	49
Priorytety operatorów	49
Operacje na ciągach znaków	50
Ćwiczenia do samodzielnego wykonania	51

Lekcja 4. Instrukcje warunkowe	51
Instrukcja if	51
Instrukcja if...else	54
Instrukcja if...else if	55
Zagnieździanie instrukcji warunkowych	56
Złożone wyrażenia warunkowe	58
Instrukcja wyboru switch	60
Operator warunkowy	62
Ćwiczenia do samodzielnego wykonania	63
Lekcja 5. Pętle	64
Pętla for	64
Pętla while	66
Pętla do...while	67
Pętla for...in	69
Jak nazywać zmienne iteracyjne?	69
Zagnieździanie pętli	70
Przerywanie pętli	72
Kontynuowanie pętli	74
Warianty pętli for	75
Ćwiczenia do samodzielnego wykonania	77
Lekcja 6. Funkcje i zasięg zmiennych	77
Jak tworzymy funkcje?	78
Argumenty funkcji	79
Zwracanie wartości przez funkcję	81
Zasięg zmiennych	82
Jak przekazywane są argumenty?	85
Pomijanie argumentów	86
Funkcje wewnętrzne	89
Ćwiczenia do samodzielnego wykonania	90
Rozdział 3. Obiekty, tablice i wyjątki	91
Lekcja 7. Standardowe obiekty i funkcje	91
Funkcje globalne	91
Właściwości globalne	100
Nieco matematyki (obiekt Math)	100
Ćwiczenia do samodzielnego wykonania	106
Lekcja 8. Tworzenie obiektów	106
Czym jest obiekt?	107
Tworzenie prostych obiektów (JSON)	107
Bezpośrednie przypisywanie właściwości	111
Odczyt i zapis danych za pomocą pętli	114
Funkcje jako właściwości obiektów	116
Ćwiczenia do samodzielnego wykonania	120
Lekcja 9. Funkcje, konstruktory i prototypy	121
Czy funkcje to obiekty?	121
Właściwości funkcji	125
Obiekt globalny	127
Konstruktory	129
Prototypy	132
Ćwiczenia do samodzielnego wykonania	138
Lekcja 10. Tablice	138
Jak tworzymy tablice?	139
Jak zapisywać i odczytywać dane?	141
Indeksy i właściwości tablic	144

Użycie pętli	145
Operacje na tablicach	149
Ćwiczenia do samodzielnego wykonania	156
Lekcja 11. Obsługa błędów i wyjątki	157
Zgłaszanie wyjątków	157
Przechwytywanie wyjątków	159
Obsługa błędów w praktyce	161
Blok finally	165
Zagnieżdżanie bloków try...catch	169
Propagacja wyjątków	169
Predefiniowane obiekty wyjątków	169
Ćwiczenia do samodzielnego wykonania	170
Rozdział 4. Współpraca z przeglądarką	171
Lekcja 12. DOM — współpraca z przeglądarką	171
Obiekty główne przeglądarki	172
Obiekt window	172
Obiekt document	181
Obiekt history	184
Obiekt location	185
Obiekt navigator	187
Lekcja 13. DOM — dostęp do elementów witryny	190
Struktura dokumentu	191
Dostęp do elementów strony WWW	193
Bezpośrednia manipulacja węzłami dokumentu	196
Tworzenie elementów strony przez skrypt	198
Usuwanie elementów strony	199
Ćwiczenia do samodzielnego wykonania	204
Lekcja 14. Zdarzenia	204
Obsługa zdarzeń	204
Ładowanie strony	207
Reagowanie na kliknięcia	211
Reagowanie na ruchy myszy	213
Dynamiczne przypisywanie procedur obsługi	215
Zdarzenia i dynamiczne elementy dokumentu	218
Ćwiczenia do samodzielnego wykonania	219
Lekcja 15. Elementy witryny	220
Elementy typu <input>	220
Przyciski	222
Pola wyboru typu checkbox	224
Pola wyboru typu radio	227
Zwykłe pola tekstowe	229
Rozszerzone pola tekstowe	232
Pola tekstowe typu password	233
Listy wyboru	235
Ćwiczenia do samodzielnego wykonania	239
Lekcja 16. Style CSS	240
Dostęp do atrybutów	240
Obiekt style	244
Właściwość className	249
Ćwiczenia do samodzielnego wykonania	251

Rozdział 5. Przetwarzanie danych	253
Lekcja 17. Operacje na ciągach znaków	253
Jak sprawdzić długość tekstu?	253
Metody formatujące ciągi znaków	256
Przetwarzanie ciągów	259
Użycie metod operujących na ciągach	265
Ćwiczenia do samodzielnego wykonania	268
Lekcja 18. Wprowadzanie danych przez użytkownika	269
Formularze	269
Sprawdzanie poprawności danych	273
Wprowadzanie danych	278
Przetwarzanie stylów	281
Ćwiczenia do samodzielnego wykonania	285
Lekcja 19. Wyrażenia regularne	286
Obiekt RegExp	286
Jak korzystać z wyrażeń?	286
Budowanie wyrażeń	290
Metody używające wyrażeń regularnych	299
Wyrażenia regularne i typ łańcuchowy	305
Wyrażenia regularne w praktyce	307
Ćwiczenia do samodzielnego wykonania	310
Lekcja 20. Cookies	311
Jak działają cookies?	312
Z czego składa się cookie?	312
Podglądarki cookies w przeglądarkach	315
Cookies i JavaScript	315
Zliczanie liczby odwiedzin	323
Ćwiczenia do samodzielnego wykonania	325
Rozdział 6. Data i czas	327
Lekcja 21. Obsługa daty i czasu	327
Obiekt Date	327
Pobieranie daty i czasu	331
Korzystanie z informacji o dacie i czasie	334
Formatowanie daty (metoda parse)	337
Data i czas w praktyce	338
Ćwiczenia do samodzielnego wykonania	341
Lekcja 22. Korzystanie z timerów	342
Timery w JavaScriptie	342
Wywołania cykliczne (interwały)	346
Symulacja działania metody setInterval	349
Zegary	350
Animacje	352
Ćwiczenia do samodzielnego wykonania	357
Skorowidz	359

Wstęp

Czym jest JavaScript?

JavaScript to skryptowy język programowania najczęściej używany do tworzenia interaktywnych stron WWW. To dzięki niemu można wyposażyc witrynę w np. dynamicznie rozwijane menu czy animowane elementy, a strony zaczynają reagować na działania użytkownika. JavaScript jest też podstawą tak popularnej ostatnimi czasy techniki AJAX, przy której użyciu strona WWW zaczyna się zachowywać jak zwykła aplikacja uruchamiana z poziomu systemu operacyjnego. Praktycznie żadna współczesna strona WWW nie obejdzie się bez JavaScriptu. Jeśli więc chcesz tworzyć WWW, musisz — oprócz HTML czy CSS — poznać też JavaScript. Dlatego właśnie powstała ta książka.

Historia JavaScriptu sięga pierwszej połowy lat 90. ubiegłego wieku. Język ten powstał jako projekt Mocha w firmie Netscape, a jego głównym projektantem był Brendan Eich. Po ukończeniu prac pierwsza dostępna wersja otrzymała nazwę LiveScript, która następnie w roku 1995 — głównie ze względów marketingowych, a za porozumieniem z firmą Sun Microsystems — została zmieniona na JavaScript. To do dnia dzisiejszego powoduje nieporozumienia i mylienie Javy z JavaScriptem. Kwestia ta zostanie wyjaśniona w dalszej części książki.

Dla kogo jest ta książka?

Oczywiście, książka przeznaczona jest dla osób, które chcą poznać JavaScript oraz nauczyć się tworzyć korzystające z tej technologii interaktywne i atrakcyjne strony internetowe. Potrzebna jest znajomość co najmniej solidnych podstaw języków HTML bądź XHTML. Przydatna będzie również wiedza na temat stosowania stylów CSS. Czytelnik powinien umieć samodzielnie tworzyć przynajmniej proste strony WWW

z użyciem jednego z tych języków. Nie jest natomiast konieczna znajomość zasad programowania (ani innych języków programowania) — zostaną one przedstawione krok po kroku w trakcie nauki.

Co będzie potrzebne do nauki?

Do nauki nie są potrzebne żadne specjalne narzędzia. Należy, oczywiście, mieć zainstalowaną przeglądarkę WWW — dowolną, współczesną, obsługującą JavaScript, np. Firefox, Internet Explorer, Konqueror, Netscape, Opera itp. Nie muszą to być najnowsze wersje, mogą być nawet produkty sprzed kilku lat, choć im nowsze, tym lepiej.

Do pisania samego kodu skryptów potrzebny też będzie dowolny, najprostszy edytor tekstowy. Najlepiej, aby miał możliwość zapisywania tekstu w kodowaniu UTF-8, a także podświetlania składni HTML i JavaScript. Można tu polecić takie produkty jak Notepad++ czy Notepad2. Można też skorzystać z dowolnego oprogramowania do tworzenia witryn, np. 1st page 2000, HotDog, kED, Pajęczek itp. Będzie to nasz warsztat pracy.

Pliki i kody źródłowe

Wszystkie listingi oraz kody źródłowe przykładów prezentowanych w książce, a także rozwiązania ćwiczeń można pobrać ze strony internetowej <http://helion.pl/ksiazki/jscpk.htm> lub bezpośrednio z serwera FTP: <ftp://ftp.helion.pl/przyklady/jscpk.zip>.

Pliki tekstowe z listingami zawierają fragmenty kodu w takiej postaci, w jakiej zostały zaprezentowane na listingach w książce (czasem są to tylko fragmenty skryptów). Oprócz tego, w osobnych katalogach znajdują się pełne wersje kodów (X)HTML i JavaScript, składające się na gotowe do uruchomienia przykłady. Każdy przykład jest więc prezentowany zarówno w postaci fragmentu omawianego w książce (o ile omawiany jest tylko fragment, a nie pełny kod), jak i w pełnej wersji gotowej do wczytania do przeglądarki.

Rozdział 1.

Pierwsze kroki

Lekcja 1. Pierwszy skrypt

Pierwsza lekcja rozpoczyna kurs. Zawarty w niej materiał to wprowadzenie w świat JavaScriptu; napiszemy pierwszy skrypt, pokazujący, jak zmusić przeglądarkę do wykonania pewnej czynności. Przyjrzymy się również różnicom między kodami HTML i XHTML. Najwięcej miejsca poświęcimy jednak sposobom umieszczania skryptów JavaScript w kodzie HTML i XHTML. Poznamy znaczniki `<script>` i `<noscript>` oraz różnice między skryptami osadzonymi a zewnętrznymi. Nieco miejsca poświęcimy również standardom JavaScriptu oraz wyjaśnimy różnice między JavaScriptem a Java.

Pierwszy skrypt

Na początku nauki najlepiej zobaczyć prosty skrypt, którego efektem działania jest wyświetlenie na ekranie pewnego napisu. Ponieważ nie chcemy komplikować tego zagadnienia, a kod ma działać zarówno w HTML, jak i XHTML, nie będziemy wyświetlać danych bezpośrednio na witrynie, ale spowodujemy, że tekst pojawi się w oknie dialogowym. A zatem musi powstać taki kod witryny, że po jej załadowaniu do przeglądarki na ekranie pojawi się nowe okno dialogowe zawierające zdefiniowany w kodzie strony tekst. Tak działająca witryna zgodna z HTML 4.01 będzie miała postać przedstawioną na listingu 1.1.

Listing 1.1. Skrypt wyświetlający okno dialogowe w HTML 4.01

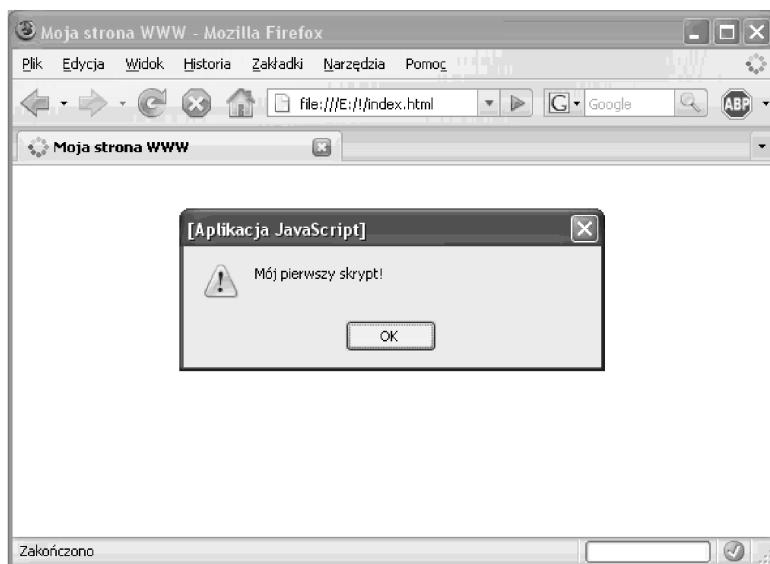
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
<script type="text/javascript">
  alert("Mój pierwszy skrypt!");
</script>
```

```
</head>
<body>
    <p>Ta strona zawiera mój pierwszy skrypt.</p>
</body>
</html>
```

To typowa strona HTML. Zawiera deklarację typu dokumentu (DOCTYPE), sekcję head, czyli nagłówek, oraz sekcję body, czyli właściwą treść dokumentu. W nagłówku znajduje się znacznik `<title>` określający tytuł i znacznik `<meta>` definiujący sposób kodowania znaków (UTF-8). W treści dokumentu widzimy natomiast znacznik `<p>` określający akapit tekstowy, który sprawia, że strona nie jest pusta, ale pojawia się na niej napis. To wszystkie elementy HTML. Jednak w sekcji head znajdujemy znacznik `<script>`, który definiuje skrypt JavaScript. Skrypt wykonuje tylko jedno zadanie — wyświetla na ekranie okno dialogowe zawierające tekst Mój pierwszy skrypt!. Jeśli zatem wczytamy taką stronę do przeglądarki, zobaczymy widok przedstawiony na rysunku 1.1.

Rysunek 1.1.

Na ekranie pojawiło się okno dialogowe zawierające zdefiniowany w skrypcie tekst



Zwrócić uwagę, że najpierw pojawiło się okno dialogowe (przy pustej stronie), a dopiero po kliknięciu znajdującego się w nim przycisku *OK* ukazał się akapit tekstowy zdefiniowany za pomocą znacznika `<p>`. Tą kwestią zajmiemy się jednak w dalszej części lekcji, najpierw zobaczymy, jak wyglądałaby taka strona w XHTML. Widać to na listingu 1.2.

Listing 1.2. Skrypt wyświetlający okno dialogowe w XHTML 1.0

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/tr/xhtml11/Dtd/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="pl">
```

```
<head>
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<title>Moja strona WWW</title>
<script type="text/javascript">
    alert("Mój pierwszy skrypt!");
</script>
</head>
<body>
    <p>Ta strona zawiera mój pierwszy skrypt.</p>
</body>
</html>
```

Jak widać, skrypt umieszczony jest dokładnie w taki sam sposób jak w poprzednim przykładzie. Również jego działanie będzie identyczne.

HTML czy XHTML?

Na pytanie, czy strony WWW tworzyć w HTML, czy XHTML, każdy twórca musi sobie odpowiedzieć sam. Ostatnia oficjalna specyfikacja HTML — 4.01 — ma już ponad 10 lat i nie da się ukryć, że jest przestarzała. XHTML jest bardziej uporządkowany i nowocześniejszy. Jednak już niedługo będzie dostępny HTML 5 (w trakcie pisania książki wciąż trwały prace nad tym standardem). To, w którym z języków wykonamy witrynę, dla użytkownika strony nie ma praktycznie żadnego znaczenia, ważne, by była zgodna ze standardami.

Przykłady skryptów w tej książce zostały napisane tak, aby działały prawidłowo zarówno w witrynach napisanych w HTML, jak i XHTML.

Umieszczanie skryptów w kodzie HTML i XHTML

Znacznik `<script>`

Dwa ostatnie przykłady pokazały, że należy użyć znacznika `<script>`, aby umieścić w kodzie HTML skrypt JavaScript. Ogólna struktura znacznika wygląda tak:

```
<script type="typ">
    tutaj treść skryptu
</script>
```

Parametr `type` określa tutaj typ języka skryptowego i jest obligatoryjny. Co prawda, jego pominięcie zwykle nie spowoduje usterki w działaniu strony (bowiem przeglądarki są na takie sytuacje przygotowane i z reguły samodzielnie potrafią określić typ skryptu), jednak jest to bardzo zła praktyka, a strona jest wtedy niezgodna ze standardami. Zatem zawsze stosujemy parametr `type` i w przypadku języka JavaScript jako jego wartość używamy ciągu znaków `text/javascript`:

```
<script type="text/javascript">
    tutaj kod skryptu
</script>
```

Uwaga: można również spotkać inne określenia typu skryptu, w tym application/javascript, text/ecmascript i application/ecmascript, które również są zazwyczaj poprawnie rozpoznawane przez przeglądarki. Jednak konsekwentnie będziemy korzystać z najpopularniejszego określenia — text/javascript.

Dawniej do określania typu skryptu stosowany był parametr language, np.:

```
<script language="JavaScript">
```

Jest on przestarzały i niezgodny z obecnymi wersjami HTML i XHTML. Nie należy go stosować, choć w sieci znajdziemy jeszcze skrypty, które z tego parametru korzystają.

Oprócz wymienionych, można również używać parametrów charset oraz defer. Pierwszy z nich pozwala określić typ kodowania znaków w skrypcie. Drugi to wskazówka dla przeglądarki, że skrypt nie modyfikuje treści strony (nie generuje dynamicznie treści strony). Znacznik `<script>` mógłby więc mieć również taką postać¹:

```
<script type="text/javascript" charset="utf-8" defer>
    tutaj kod skryptu
</script>
```

Tych parametrów nie będziemy jednak stosować, ograniczając się jedynie do type.

Skrypty osadzone

Skrypty osadzone to takie, których treść jest wpleciona w kod HTML. Wyglądają one tak, jak na listingach 1.1 i 1.2. Należy je stosować, jeśli kod skryptu nie jest zbyt długi i skomplikowany. Ponieważ w dzisiejszych czasach, ze względu na stosowanie coraz to nowych i bardziej zaawansowanych technik (np. takich jak AJAX), rozmiar i skomplikowanie skryptów wciąż rosną, coraz częściej rozdziela się kody (X)HTML oraz JavaScript i umieszcza w różnych plikach. Mamy wtedy do czynienia ze skryptami zewnętrznymi.

Skrypty zewnętrzne

Jeśli treść skryptu JavaScript ma się znaleźć w innym pliku niż kod HTML witryny, należy również zastosować znacznik `<script>`, ale wyposażyć go w parametr src wskazujący adres URL pliku ze skryptem. Znacznik ten będzie miał wtedy postać:

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

Adres URL może mieć postać bezwzględną, np.:

<http://mojadomena.com/skrypty/skrypt.js>

¹ W przypadku XHTML i parametru defer należało użyć ciągu `<script type="text/javascript" charset="utf-8" defer="defer">`.

Lub, jeśli skrypt znajduje się na tym samym serwerze, co kod witryny, względna:
`/skrypty/skrypt.js`

Jeżeli plik ze skryptem znajduje się w tym samym katalogu, co plik z kodem strony, wystarczy podać samą nazwę, bez określania ścieżki dostępu:

`skrypt.js`

Znacznik `<script>` może więc przyjąć taką postać:

```
<script type="text/javascript" src="skrypt.js">  
</script>
```

Przyjmuje się przy tym, że plik z kodem skryptu powinien mieć rozszerzenie `.js`, choć nie jest to obligatoryjne i można użyć dowolnego innego. W książce będzie konsekwentnie stosowane rozszerzenie `.js`.

Oczywiście, w takim przypadku cała zawartość skryptu powinna znaleźć się w pliku o nazwie `skrypt.js`, przy czym, uwaga, w treści tego pliku nie stosujemy już znacznika `<script>`, ale wpisujemy sam kod skryptu. W przykładzie z listingu 1.1 całą zawartością takiego pliku byłby tylko jeden wiersz o treści:

```
alert("Mój pierwszy skrypt!");
```

Należy również pamiętać o właściwym kodowaniu znaków w pliku ze skryptem. Powinno ono być zgodne z zadeklarowanym w znaczniku `<script>` lub zdefiniowanym w sekcji head dokumentu HTML. Jeśli więc stronę zapisujemy w kodowaniu ISO-8859-2, skrypt najlepiej również zapisać w tym kodowaniu. To samo dotyczy standardu UTF-8 czy Windows-1250.

Gdzie umieszczać skrypty?

Skrypt można umieścić w sekcji head lub sekcji body. Jaka jest różnica? Jeśli skrypt znajduje się pomiędzy znacznikami `<head>` i `</head>`, zostanie wykonany jeszcze przed załadowaniem właściwej treści strony. Właśnie z taką sytuacją mieliśmy do czynienia w przykładach z listingu 1.1 i 1.2. Najpierw pojawiło się wyświetlane przez skrypt okno dialogowe, a dopiero potem właściwa treść witryny.

Jeśli jednak skrypt umieścimy w sekcji `<body>`, najpierw zostanie wyświetlona część strony znajdująca się przed skryptem, potem wykonany skrypt, a następnie wyświetlona część strony znajdująca się za skryptem. Rozważmy skrypt widoczny na listingu 1.3.

Listing 1.3. Skrypt w treści strony w sekcji body

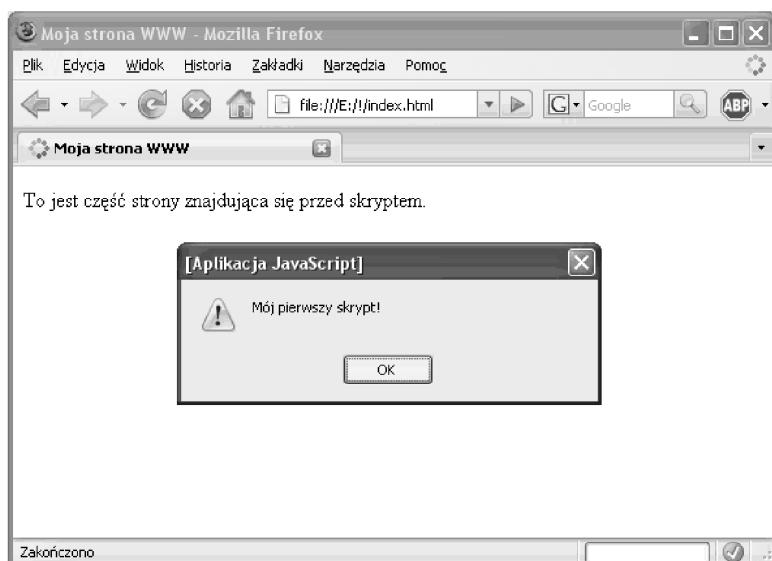
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">  
<html>  
<head>  
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">  
  <title>Moja strona WWW</title>  
</head>
```

```
<body>
<p>To jest część strony znajdująca się przed skryptem.</p>
<script type="text/javascript">
    alert("Mój pierwszy skrypt!");
</script>
<p>Ta część strony znajduje się za skryptem.</p>
</body>
</html>
```

Tym razem skrypt rozdziela dwa akapity tekstowe zdefiniowane za pomocą znaczników <p>. Zgodnie z tym, co zostało napisane powyżej, na ekranie najpierw powinien pojawić się tekst pierwszego akapitu, a następnie okno dialogowe generowane przez skrypt. Ta faza jest widoczna na rysunku 1.2. Dopiero po zamknięciu okna dialogowego za pomocą przycisku *OK* na ekranie pojawi się treść drugiego akapitu. Ten etap jest widoczny na rysunku 1.3.

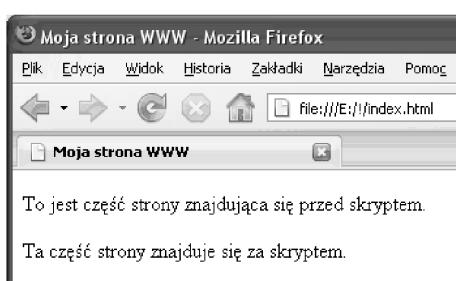
Rysunek 1.2.

Wyświetlanie strony zostało zatrzymane przez okno generowane za pomocą skryptu



Rysunek 1.3.

Po zamknięciu okna dialogowego pojawiła się dalsza część witryny



Jakie to ma znaczenie praktyczne? Otóż, zaleca się, aby — o ile to możliwe — umieścić skrypty w sekcji head. Dzięki temu są one wczytywane i, ewentualnie, wykonywane przed załadowaniem strony. Założymy, że skrypt ma odpowiedzieć na jakieś działanie użytkownika, np. kliknięcie, jednak wcześniej potrzebuje przygotować pewne dane.

Jeśli najpierw zostanie wczytana strona i użytkownik zdąży wykonać kliknięcie przed przygotowaniem danych przez skrypt, wystąpi błąd. Często też korzystamy z bibliotek (zbiorów skryptów) dostarczanych przez innych programistów i one powinny być wczytane przed przetworzeniem witryny. Odwołania do skryptów w sekcji body są więc jak najbardziej na miejscu.

Niejednokrotnie zdarza się też, że skrypt operuje na jakimś elemencie witryny. Tu z kolei, jeśli zostanie wykonany przed załadowaniem i przetworzeniem przez przeglądarkę kodu HTML strony, tego elementu jeszcze nie będzie i również wystąpi błąd. W takiej sytuacji lepiej byłoby umieścić kod skryptu za instrukcjami HTML tworzącymi dany element.

Obu opisanych problemów można też uniknąć, stosując odpowiednie techniki programistyczne. Przyjmijmy jednak ogólną zasadę, że — o ile to możliwe — staramy się umieszczać odwołania do skryptów w sekcji head oraz korzystamy raczej ze skryptów zewnętrznych (chyba że są to tylko małe fragmenty kodu).

Znacznik <noscript>

Znacznik <noscript> pozwala zdefiniować treść alternatywną dla przeglądarek, które nie obsługują skryptów. Powinien się znaleźć tuż za znacznikiem <script>. Ogólnie taka konstrukcja ma postać:

```
<script type="typ">
    tutaj treść skryptu
</script>
<noscript>
    tutaj treść alternatywna
</noscript>
```

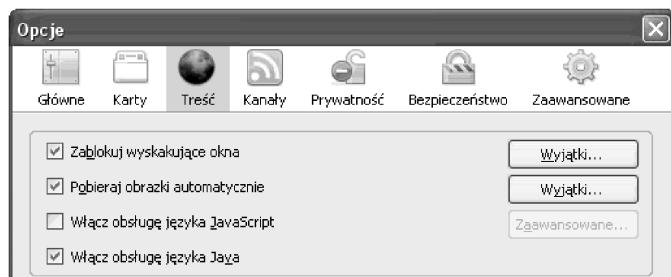
Jeśli przeglądarka rozpoznaje znacznik <script>, ale nie wykonuje skryptów, pominie kod znajdujący się między znacznikami <script> oraz </script> i wyświetli zawartość umieszczoną pomiędzy znacznikami <noscript> i </noscript>. W praktyce wyglądałoby to tak, jak na listingu 1.4.

Listing 1.4. Użycie znacznika <noscript>

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Moja strona WWW</title>
</head>
<body>
    <script type="text/javascript">
        alert("Mój pierwszy skrypt!");
    </script>
    <noscript>
        <p>Twoja przeglądarka nie obsługuje skryptów.</p>
    </noscript>
</body>
</html>
```

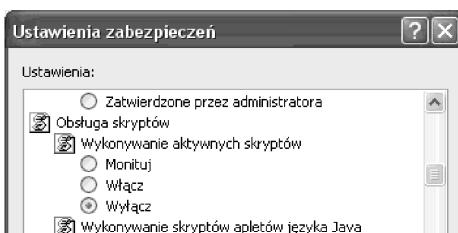
Działanie kodu najłatwiej sprawdzić, włączając i wyłączając obsługę skryptów w przeglądarce. Wyłączanie skryptów przebiega odmiennie w różnych produktach. W Firefoxach 2 i 3 z menu *Narzędzia* należy wybrać pozycję *Opcje*, a następnie kliknąć ikonę *Treść* i usunąć zaznaczenie pola *Włącz obsługę języka JavaScript* (rysunek 1.4).

Rysunek 1.4.
*Wyłączanie obsługi
JavaScriptu
w przeglądarce
Firefox 2*



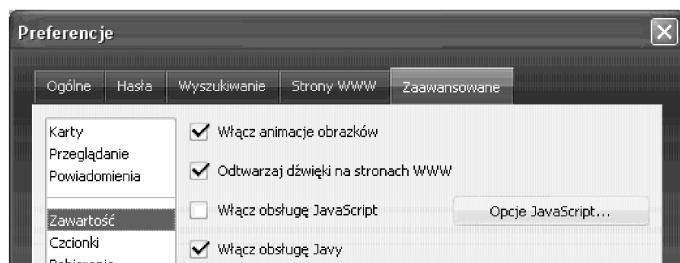
W Internet Explorerach 6 i 7 z menu *Narzędzia* wybieramy pozycję *Opcje internetowe*, a następnie klikamy zakładkę *Zabezpieczenia*, a na niej przycisk *Poziom niestandardowy*. Na ekranie pojawi się okno *Ustawianie zabezpieczeń* (rysunek 1.5), w którym odszukujemy sekcję *Obsługa skryptów* i w opcji *Wykonywanie aktywnych skryptów* zaznaczamy opcję *Wyłącz*. Niezbędne też będzie ponowne uruchomienie przeglądarki.

Rysunek 1.5.
*Wyłączanie skryptów
w przeglądarce
Internet Explorer 6*



W Operze 9 z menu *Narzędzia* wybieramy pozycję *Preferencje*. W oknie *Preferencje* (rysunek 1.6) klikamy zakładkę *Zaawansowane*, a następnie opcję *Zawartość*, w której usuwamy zaznaczenie pola wyboru *Włącz obsługę JavaScript*.

Rysunek 1.6.
*Wyłączanie skryptów
w przeglądarce
Opera 9*



Jeśli zatem wyłączymy JavaScript w przeglądarce i wczytamy do niej kod z listingu 1.4, skrypt nie zostanie wykonany, ale zamiast niego pojawi się zawartość znacznika <noscript>, czyli akapit tekstowy z informacją (rysunek 1.7). Oczywiście, kiedy włączymy JavaScript, kod strony zachowią się tak, jak w przykładach z listingów 1.1 i 1.2, a więc zostanie wykonany skrypt, a pominięta zawartość znacznika <noscript>.

Rysunek 1.7.

Po wyłączeniu skryptów została wyświetlona alternatywna zawartość



Standardy JavaScript

JavaScript w kilku pierwszych latach swojego istnienia rozwijał się dosyć burzliwie i nie był poddany standaryzacji. Dlatego też rozwinęło się kilka dialektów i wersji tego języka. Stąd daje się zauważać pewne zamieszanie panujące w kwestii nazewnictwa oraz istniejących wersji tego języka. Najczęściej spotkamy się z nazwami JavaScript, JScript i ECMAScript, ale również ActionScript, CriScript czy EJScript.

Co to oznacza? Otóż, gdy kolejne lata rozwoju powodowały powstawanie coraz to mniej zgodnych ze sobą wersji, niezbędna stała się standaryzacja. Stworzeniem takiego standardu zajęła się międzynarodowa organizacja ECMA (skrót pochodzi od dawnej nazwy — *European Computer Manufacturers Association*). W ten sposób powstał ECMAScript. Zatem nazwa ECMAScript określa standard języka. Z kolei JavaScript czy JScript to implementacje tego standardu występujące w konkretnych produktach, najczęściej przeglądarcach — tą tematyką zajmujemy się przecież w książce. W Firefoksie i Operze stosowana jest implementacja nazywana JavaScript, a w Internet Explorerze — implementacja o nazwie JScript.

Zależności pomiędzy poszczególnymi wersjami zostały zaprezentowane w tabeli 1.1.

Nie należy się jednak tą mnogością przejmować. W każdej współczesnej, popularnej przeglądarce mamy do czynienia z praktycznie tym samym językiem i będziemy go nazywać po prostu JavaScript. Wszystkie podstawowe konstrukcje (a także większość bardzo zaawansowanych, których i tak nie będziemy stosować) są takie same i zachowują się tak samo. Nie spotkamy się więc z problemami dotyczącymi implementacji tego języka².

JavaScript i Java

Java i JavaScript to dwa zupełnie różne języki, a także dwie zupełnie odmienne technologie. Zatem JavaScript to NIE SĄ skrypty Java! Nie ma czegoś takiego jak skrypty Java. To, że te dwie nazwy są do siebie podobne, znaczy tylko tyle, że... są do siebie podobne. I nic więcej. Zwracam uwagę na tę kwestię, bowiem jest to często spotykany błąd. O ile mają do niego pełne prawo osoby początkujące, zadające pytania na forach

² Jak przekonamy się w dalszej części książki, problemem jest zupełnie co innego — różnice w sposobach interpretacji strony WWW w różnych przeglądarcach. Jednak z tym poradzimy sobie bez trudu.

Tabela 1.1. Zależności pomiędzy różnymi wersjami JavaScriptu

ECMAScript	JavaScript	JScript	Przeglądarki
–	1.0	1.1	Netscape 2.0, Internet Explorer 3.0
–	1.1	2.0	Netscape 3.0, Internet Explorer 3.0
–	1.2	–	Netscape 4.0 – 4.05
v1	1.3	3.0	Netscape 4.06 – 4.7, Internet Explorer 4
–	1.4	4.0	Brak implementacji w przeglądarkach
–	–	5.0, 5.1	Internet Explorer 5.0, 5.1
v3	1.5	5.5	Netscape 6, Firefox 1, Internet Explorer 5.5
v3	–	5.6	Internet Explorer 6
v3	1.6	–	Firefox 1.5
v3	1.7	–	Firefox 2
v3	1.8	–	Firefox 3
v4	2.0	–	Dotychczas brak implementacji w przeglądarkach ³

internetowych czy grupach news, to już autorzy umieszczający na swoich stronach zbiory skryptów mogliby zadać sobie nieco więcej trudu (a nie tylko kopiować gotowe rozwiązania) i sprawdzić, co właściwie prezentują gościom odwiedzającym ich witryny. Co gorsza, ten nieprawidłowy termin spotyka się również w popularnych pismach komputerowych, co jest już zupełnie niezrozumiałe.

Owszem, między Java a JavaScriptem występują pewne podobieństwa w składni typowych konstrukcji, np. pętli, instrukcji warunkowych itp., ale są one typowe dla wszystkich języków programowania, których składnia wywodzi się z języka C. Podkreślam więc raz jeszcze, że nazwa języka brzmi JavaScript i mówimy o skryptach JavaScript. Osobom, które chciałby poznać właściwy język Java (i przy okazji przekonać się, że jest to faktycznie coś zupełnie innego niż JavaScript), można polecić książki, które ukazały się nakładem wydawnictwa Helion: *Java. Ćwiczenia praktyczne. Wydanie II* (<http://helion.pl/ksiazki/cwjav2.htm>) i *Java. Praktyczny kurs7. Wydanie II* (<http://helion.pl/ksiazki/pkjav2.htm>).

³ W trakcie pisania książki czwarta wersja ECMAScript była dopiero w ostatniej fazie przygotowań. Żadna przeglądarka nie mogła więc jej implementować.

Rozdział 2.

Instrukcje języka

Lekcja 2.

Instrukcje, zmienne i typy danych

Lekcja 2. rozpoczyna rozdział 2., w którym omówione zostaną wszystkie podstawowe konstrukcje języka JavaScript. Dowiemy się, czym są instrukcje, poznamy pojęcie zmiennej i typu danych, a także przyjrzymy się konkretnym typom danych występującym w JavaScripcie. Nie pominiemy tematu komentarzy, za których pomocą można opisywać słowami kody skryptów, a także ukryć treść skryptu przed przeglądarką. Przedstawionych zostanie również kilka wskazówek dotyczących sposobu zapisu skryptów i ich interpretacji przez przeglądarkę. Zaczniemy jednak od przygotowania kodów HTML i XHTML wspólnych dla większości przykładów z tego rozdziału. To bardzo ułatwi dalszą pracę.

Struktura kodów HTML i XHTML w tym rozdziale

Cały rozdział 2. poświęcony jest podstawowym konstrukcjom języka JavaScript. Ich poznanie jest niezbędne, aby można było pisać skrypty. Oczywiście, najłatwiej uczyć się na przykładach, których efekt działania można zobaczyć na witrynie. W tym celu trzeba zmusić skrypty do wyświetlania danych. Co prawda, już w lekcji 1. poznaliśmy metodę wyświetlania okna dialogowego, jednak używanie jej w dalszych przykładach nie byłoby zbyt wygodne. Najlepiej bowiem, aby wyniki działania skryptów pojawiały się wprost na stronie. Niestety, najprostsze możliwe rozwiązanie nie będzie działało jednocześnie w HTML i XHTML (jest to instrukcja `document.write`, która zostanie opisana w dalszej części lekcji). Trzeba więc przygotować rozwiązanie uniwersalne, które dodatkowo pozwoli usunąć z listingów powtarzające się elementy. Struktura kodów HTML i XHTML we wszystkich przykładach będzie taka sama. Kod HTML został przedstawiony na listingu 2.1.

Listing 2.1. Kod HTML dla przykładów z 2. rozdziału

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
</head>
<body>
<div id="dataDiv">
</div>
<script type="text/javascript" src="skrypt.js">
</script>
<noscript>
<p>Twoja przeglądarka nie obsługuje skryptów.</p>
</noscript>
</body>
</html>
```

To typowa strona, w której w sekcji body za pomocą znacznika `<div>` została zdefiniowana warstwa o identyfikatorze dataDiv. Identyfikator pozwoli odwoływać się do warstwy w skryptach. Znacznik `<script>` został umieszczony za warstwą, tak aby zdefiniowany przezeń skrypt został wykonany dopiero po tym, jak przeglądarka utworzy warstwę. Jest to skrypt zewnętrzny umieszczony w pliku o nazwie *skrypt.js*. Plik ten powinien zostać umieszczony w tym samym katalogu, co plik z kodem HTML strony.

Kod XHTML będzie wyglądał bardzo podobnie. Został zaprezentowany na listingu 2.2 i pełni takie same funkcje. Różni się jedynie sposobem definicji nagłówków.

Listing 2.2. Kod XHTML dla przykładów z 2. rozdziału

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/tr/xhtml11/Dtd/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="pl">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Moja strona WWW</title>
</head>
<body>
<div id="dataDiv">
</div>
<script type="text/javascript" src="skrypt.js">
</script>
<noscript>
<p>Twoja przeglądarka nie obsługuje skryptów.</p>
</noscript>
</body>
</html>
```

Pozostaje określić, jaka będzie ogólna struktura kodów JavaScript. Otóż, będą miały postać przedstawioną na listingu 2.3.

Listing 2.3. Ogólna struktura kodów JavaScript z 2. rozdziału

```
// Tutaj będziemy umieszczać kod generujący dane.  
// Wszystkie dane, które mają się pojawić na stronie,  
// należy zapisać w zmiennej o nazwie str.  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Jak widać, znaczenie mają tutaj dwa ostatnie wiersze. Co dokładnie oznaczają, dowiemy się dopiero w rozdziale 3. W tej chwili przyjmijmy, że powodują wyświetlenie na warstwie `dataDiv` zawartości zmiennej o nazwie `str`, czyli to, co jest zapisane w `str`, pojawi się na witrynie. O tym, czym są zmienne i jak się nimi posługiwać, dowiemy się jeszcze w tej lekcji.

W dalszej części rozdziału kody HTML i XHTML nie będą już publikowane, gdyż w każdym przykładzie użyjemy tych samych. Na listingach prezentowana będzie jedynie treść skryptów JavaScript.

A zatem podsumujmy. Jeśli korzystamy z HTML, zapisujemy kod z listingu 2.1 w pliku `index.html` (lub o innej dowolnie wybranej nazwie), natomiast kod z listingu 2.3 (i kolejnych) — w pliku `skrypt.js`. Następnie wczytujemy do przeglądarki plik `index.html` i wtedy skrypt zostanie wykonany.

Jeżeli używamy XHTML, postępujemy analogicznie, z tą różnicą, że kod z listingu 2.2 zapisujemy w pliku `index.xhtml` (lub o innej dowolnie wybranej nazwie).

Osoby, które nie chcą korzystać ze skryptów zewnętrznych i zamierzają cały kod, czyli i HTML (XHTML), i JavaScript, zapisywać w jednym pliku, również mogą to zrobić. Wtedy znacznik `<script>` z listingów 2.1 i 2.2 będzie miał postać:

```
<script type="text/javascript">  
    Tutaj wpisz treść skryptu.  
</script>
```

Czym są instrukcje?

Instrukcję możemy potraktować jak polecenie do wykonania. Skrypt napisany w języku JavaScript to w istocie ciąg instrukcji, a zatem ciąg poleceń. Polecenia są interpretowane i wykonywane przez przeglądarkę (a ściślej, przez tzw. interpreter, inaczej — aparat wykonawczy JavaScriptu). Możemy zatem nakazać wyświetlenie okna dialogowego, umieścić jakąś treść na stronie WWW, zmodyfikować zawartość warstwy czy innego elementu witryny itp.

Co ważne, każdą instrukcję powinniśmy zakończyć znakiem średnika. To informacja dla interpretera, że jest to koniec instrukcji. Jak rozpoznać, co jest instrukcją, a co nie, i kiedy należy stawić średnik? Ta wiedza przychodzi sama po wykonaniu kliku przykładów, więc po przeczytaniu tego rozdziału na pewno nikt nie będzie miał wątpliwości.

Najlepiej wykonajmy od razu pierwszy przykład i przetestujmy przygotowane na początek tej lekcji kody. Chcielibyśmy na ekranie wyświetlić pewien tekst. Niech będzie to: „Umiem wyświetlać napisy!”. Jak powinien wyglądać kod skryptu? Tak jak na listingu 2.4.

Listing 2.4. Skrypt wyświetlający napis na witrynie

```
var str = "Umiem wyświetlać napisy!";
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

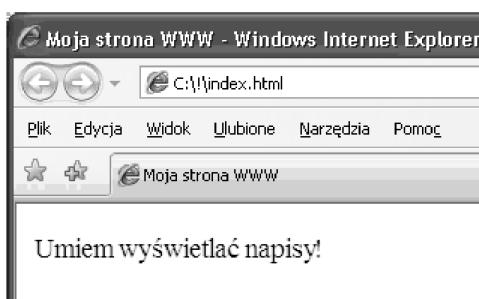
Widzimy tu trzy wiersze programu i są to trzy instrukcje. Każda z nich jest zakończona znakiem średnika. Wiemy już, że dwie ostatnie powodują wyświetlenie na stronie dataDiv tego, co znajduje się w zmiennej o nazwie str. Taka zmienna powstaje w pierwszej instrukcji:

```
var str = "Umiem wyświetlać napisy!";
```

A zatem, najpierw umieszczamy w zmiennej str napis, a następnie wyświetlamy go na ekranie. Dlatego po wczytaniu do przeglądarki witryny zawierającej taki skrypt, zobaczymy widok przedstawiony na rysunku 2.1. Czym jest owa tajemnicza zmienna? Dowiemy się tego w kolejnej części lekcji.

Rysunek 2.1.

Efekt działania skryptu wyświetlającego napis



Co to jest zmienna?

Zmienna to miejsce w skrypcie, w którym można przechowywać dane, czyli liczby, napisy itp. Miejsce to oznaczone jest nazwą. A zatem każda zmienna ma swoją nazwę, która pozwala na jej jednoznaczną identyfikację w treści skryptu, inaczej mówiąc, „w kodzie”. Zmienna charakteryzuje się również typem, który określa rodzaj danych, jakie przechowuje. W celu utworzenia zmiennej należy ją zadeklarować, tzn. podać nazwę oraz początkową wartość. Można to zrobić na dwa sposoby, schematycznie:

```
var nazwa_zmiennej = wartość;
```

lub:

```
nazwa_zmiennej = wartość;
```

Na potrzeby najbliższych lekcji możemy uznać, że oba te sposoby są równoważne. Różnice omówimy dopiero w dalszej części książki.

Przykład deklaracji zmiennej już poznaliśmy. Przypomnijmy sobie wiersz z listingu 2.4:

```
var str = "Umiem wyświetlać napisy!";
```

To nic innego jak deklaracja zmiennej o nazwie str i przypisanie jej ciągu znaków (napisu) Umiem wyświetlać napisy!. Należy tu zwrócić uwagę, że ciąg ten został ujęty w znaki cudzysłowu. Jest to niezbędne. To informacja dla przeglądarki¹, gdzie zaczyna się i kończy napis umieszczony przez nas w kodzie skryptu; nie potraktuje go więc np. jak nieznaną instrukcję.

Oczywiście, zmiennej można przypisywać nie tylko ciągi znaków, ale również np. wartości liczbowe. Oto przykład:

```
var wartosc = 24;
```

Tym razem nie użyliśmy cudzysłowu. Jest to właściwe postępowanie, gdyż zmiennej wartosc chcieliśmy przypisać wartość liczbową, a nie napis.

Nie możemy też pominąć kwestii nazewnictwa zmiennych. Otóż, nazwa może zawierać litery, cyfry, znak dolara (\$) i znak podkreślenia. Może również składać się ze znaków diakrytycznych (czyli np. polskich liter ą, ź czy ć). Wolno stosować zarówno wielkie, jak i małe litery, ale są one rozróżniane, co oznacza, że przykładowo liczba i Liczba to nazwy dwóch różnych zmiennych. Nazwa zmiennej nie może zaczynać się od cyfry. W praktyce, kiedy nazywamy zmienną, warto użyć terminu, który określa jej rolę w skrypcie. Znaku dolara najlepiej nie używać w ogóle.

Typy danych w JavaScript

Typ danych to po prostu określenie rodzaju danych. Przykładowo typ całkowitoliczbowy określa liczby całkowite. W poprzedniej części lekcji padło stwierdzenie, że zmienna charakteryzuje się również typem. Chodzi właśnie o typ, określający, jaki rodzaj danych przechowuje dana zmienna. Przyjrzyjmy się więc typom danych występującym w JavaScript. Możemy je podzielić na następujące rodzaje:

- ◆ typ liczbowy,
- ◆ typ łańcuchowy,
- ◆ typ logiczny,
- ◆ typ obiektowy,
- ◆ typy specjalne.

¹ Oczywiście, chodzi o zawarty w przeglądarce aparat wykonawczy przetwarzający skrypty. Przyjmujemy jednak taką uproszczoną terminologię.

Typ liczbowy

Typ liczbowy służy do reprezentacji liczb, przy czym nie ma występującego w klasycznych językach programowania rozróżnienia na typy całkowitoliczbowe i rzeczywiste (zmiennopozycyjne). Liczby zapisywane są za pomocą literałów (stałych napisowych, z ang. *string constant, literal constant*) liczbowych, czyli ciągów znaków składających się na liczbę, np. 24 (umieszczone w kodzie skryptu tekst 24 to dwa znaki, dwójka i czwórka, które razem stanowią literał — stałą napisową — interpretowany jako liczba 24). Obowiązują przy tym następujące zasady.

- ◆ Jeżeli ciąg cyfr nie jest poprzedzony żadnym znakiem lub jest poprzedzony znakiem +, reprezentuje wartość dodatnią, jeżeli natomiast jest poprzedzony znakiem -, reprezentuje wartość ujemną.
- ◆ Jeżeli literał rozpoczyna się od cyfry 0, jest traktowany jako wartość ósemkowa.
- ◆ Jeżeli literał rozpoczyna się od ciągu znaków 0x, jest traktowany jako wartość szesnastkowa (heksadecymalna). W zapisie wartości szesnastkowych mogą być wykorzystywane zarówno małe, jak i wielkie litery alfabetu, od A do F.
- ◆ Literaly mogą być zapisywane w notacji wykładniczej, w postaci X.YeZ, gdzie X to część całkowita, Y — część dziesiętna, natomiast Z to wykładnik potęgi liczby 10. Zapis taki oznacza to samo, co X.Y * 10^Z.

Przykłady literałów:

- 123 — dodatnia całkowita wartość dziesiętna 123,
- 123 — ujemna całkowita wartość dziesiętna -123,
- 012 — dodatnia całkowita wartość ósemkowa równa 10 w systemie dziesiętnym,
- 024 — ujemna całkowita wartość ósemkowa równa 20 w systemie dziesiętnym,
- 0xFF — dodatnia całkowita wartość szesnastkowa równa 255 w systemie dziesiętnym,
- 0x0f — ujemna całkowita wartość szesnastkowa równa -15 w systemie dziesiętnym,
- 1.1 — dodatnia wartość rzeczywista 1.1,
- 1.1 — ujemna wartość rzeczywista -1.1,
- 0.1E2 — dodatnia wartość rzeczywista równa 10,
- 1.0E-2 — dodatnia wartość rzeczywista równa 0.01.

Typ łańcuchowy

Typ łańcuchowy (inaczej, typ *string*) służy do reprezentacji ciągów znaków (napisów). Ciagi te (inaczej, stałe napisowe) powinny być ujęte w znaki cudzysłowu, aczkolwiek dopuszczalne jest również wykorzystanie znaków apostrofu. Przykładowy ciąg ma postać:

"abcdefg"

Mogą też zawierać sekwencje znaków specjalnych przedstawione w tabeli 2.1.

Tabela 2.1. Sekwencje znaków specjalnych

Sekwencja	Znaczenie
\b	backspace (ang. <i>backspace</i>)
\n	nowa linia (ang. <i>new line</i>)
\r	powrót karetki (ang. <i>carriage return</i>)
\f	nowa strona (ang. <i>form feed</i>)
\t	tabulacja (ang. <i>horizontal tab</i>)
\"	cudzysłów (ang. <i>double quote</i>)
\'	apostrof (ang. <i>single quote</i>)
\\	lewy ukośnik (ang. <i>backslash</i>)

Gdzie mogą się przydać sekwencje znaków specjalnych? Oczywiście, wszędzie tam, gdzie chcemy danego znaku użyć. Co np. zrobić, jeżeli w napisie chcemy użyć znaku cudzysłowu? Założmy, że chcielibyśmy przypisać zmiennej str napis: To jest cytat: "Być albo nie być...". Nie możemy użyć zwykłych znaków cudzysłowu, gdyż wtedy cała konstrukcja miałaby postać:

```
var str = "To jest cytat: "Być albo nie być...".";
```

Niestety, tego przeglądarka nie będzie w stanie poprawnie zinterpretować, bo nie wie, które znaki cudzysłowu czego dotyczą. Potraktuje to jako ciąg znaków To jest cytat:, za którym występują nieznane jej instrukcje Być albo nie być..., a za nimi kolejny ciąg znaków zawierający jedynie kropkę. Przypisanie czegoś takiego zmiennej nie jest możliwe.

A zatem, jeśli w napisie (stałej znakowej, napisowej) miałyby się pojawić jakiekolwiek znaki specjalne, powinniśmy zamiast nich użyć sekwencji znaków specjalnych. Aby więc na witrynie pojawił się wspomniany napis, należy wykorzystać skrypt z listingu 2.5. Po jego użyciu na ekranie zobaczymy widok przedstawiony na rysunku 2.2.

Listing 2.5. Użycie sekwencji znaków specjalnych

```
var str = "To jest cytat: \"Być albo nie być...\".";
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Rysunek 2.2.

Na stronie pojawił się prawidłowy cytat



Typ logiczny

Typ logiczny (*boolean*) pozwala na określenie dwóch wartości logicznych: *prawda* i *falsz*. Wartość *prawda* jest w JavaScriptie reprezentowana przez słowo `true`, natomiast wartość *falsz* przez `false`. Wartości tego typu są używane przy konstruowaniu wyrażeń logicznych, porównywaniu danych, wskazywaniu, czy dana operacja zakończyła się sukcesem itp.

Typ obiektowy

Typ obiektowy służy do reprezentacji obiektów. Nie ma specjalnego słowa kluczowego oznaczającego ten typ. Najczęściej wykorzystuje się obiekty wbudowane oraz udostępniane przez przeglądarkę. Zajmiemy się nimi bliżej w kolejnych rozdziałach.

Typy specjalne

Mozemy wyróżnić dwa rodzaje typów specjalnych: `null` i `undefined`. Choć podobne, nie są tożsame. Otóż, `null` określa wartość pustą (czy też brak wartości) i najczęściej używany jest podczas korzystania z obiektów i programowania obiektowego. W tym kontekście jeśli jakiś typ programistyczny (zmienna, obiekt, właściwość itd.) ma wartość `null`, oznacza to, że jest pusty. Dokładniej wyjaśnimy to w rozdziale 3.

Z kolei typ `undefined` określa wartość niezdefiniowaną. W tym kontekście wartość `undefined` ma niezainicjalizowaną zmienną, zmienną, której jawnie przypisano wartość `undefined`, bądź też nieistniejąca właściwość obiektu (ten ostatni przypadek również omówimy w rozdziale 3.).

Zmienne a typy danych

Jak łatwo było zauważyć w części dotyczącej tworzenia zmiennych, w trakcie deklaracji nigdzie nie podaje się typu zmiennej. Jest tak dlatego, że w JavaScriptie typ zmiennej jest określany przez wartość jej przypisywaną. Jeśli zatem przypiszemy zmiennej wartość numeryczną (liczbową), będzie ona typu liczbowego, jeśli zaś łańcuch znaków, będzie typu łańcuchowego itd. Co więcej, typ danych nie jest przypisywany zmiennej na stałe i może się zmieniać w trakcie działania skryptu. Jeżeli w skrypcie znajdą się instrukcje:

```
wartosc = 123;  
wartosc = "abc";
```

to pierwsza z nich spowoduje utworzenie zmiennej o nazwie `wartosc` i przypisanie jej wartości liczbowej 123 — w związku z tym, typem zmiennej będzie typ liczbowy — natomiast druga instrukcja spowoduje zmianę zarówno wartości zmiennej, jak i jej typu. Będzie ona wtedy zawierała ciąg znaków oraz będzie typu łańcuchowego. Nie musimy jednak na początku nauki zajmować się bliżej zmianą typów zmiennych, gdyż — jak widać — dzieje się to automatycznie.

Wróćmy teraz do deklaracji zmiennych. W sposobie bez użycia słowa var, np.:

zmienna = 123;

zmiennej musi być przypisana wartość początkowa. Niedopuszczalne byłoby użycie konstrukcji:

zmienna;

Gdy jednak użyjemy słowa var, można zadeklarować zmienną bez przypisywania jej wartości. Jaka więc będzie wartość takiej zmiennej? Będzie to undefined. A zatem zmienią zadeklarowana ze słowem var, której nie przypisano żadnej wartości początkowej, będzie miała wartość specjalną undefined. Obrazuje to przykład widoczny na listingu 2.6.

Listing 2.6. Zmienna bez przypisanej wartości początkowej

```
var str;  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Oczywiście, nic nie stoi na przeszkodzie, aby takiej zmiennej przypisać wartość w dalszej części kodu. Jej typ zmieni się wtedy na typ przypisywanej wartości. Taka operacja jest wykonywana w kodzie widocznym na listingu 2.7.

Listing 2.7. Zmiana wartości zmiennej typu undefined

```
var str;  
str = "przykładowy ciąg znaków";  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Zmiennej można też bezpośrednio przypisać wartość specjalną undefined:

str = undefined;

Znaczniki HTML

Do tej pory na stronie wyświetliśmy jedynie czysty tekst. Jednak nic nie stoi na przeszkodzie, aby dodać do niego znaczniki HTML. Tak więc skrypt z powodzeniem może generować dynamicznie elementy witryny. I tak możemy formatować tekst za pomocą znaczników , <i>, <sup> czy
. Przykład użycia znaczników formatujących można zobaczyć na listingu 2.8, a efekt jego działania — na rysunku 2.3.

Listing 2.8. Formatowanie tekstu

```
var str = "<b>Tekst pogrubiony</b><br />" +  
        "<i>Kursywa</i><br />" +  
        "Indeks<sup>górnny</sup><br />" +  
        "Indeks<sub>dolny</sub><br />";  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Rysunek 2.3.
Efekt działania znaczników formatujących tekst



Zwróćmy przy tym uwagę, jak została zapisana instrukcja przypisująca wartość zmiennej str. Składa się ona z czterech wierszy. Dzięki temu jest czytelniejsza. Jednak pomiędzy poszczególnymi ciągami znaków ujętymi w znaki cudzysłowu znajdują się znaki +. W JavaScriptie oznaczają one, tak jak w matematyce, dodawanie, a więc w rzeczywistości nastąpiło tu połączenie czterech różnych łańcuchów znakowych. Więcej o tym dowiemy się w lekcji 3. Oczywiście, można cały napis zapisać w jednym wierszu. Miałby wtedy postać:

```
var str = "<b>Tekst pogrubiony</b><br /><i>Kursywa</i><br />Indeks<sup>górnny</sup><br />Indeks<sub>dolny</sub><br />";
```

Efekt działania byłby taki sam.

Nie wolno natomiast użyć, dopuszczalnej w niektórych innych językach, konstrukcji w postaci:

```
var str = "<b>Tekst pogrubiony</b><br />
<i>Kursywa</i><br />
Indeks<sup>górnny</sup><br />
Indeks<sub>dolny</sub><br />";
```

gdź spowodowałby ona powstanie błędu i skrypt nie zostałby wykonany (po wczytaniu witryny do przeglądarki ukazałaby się pusta strona). Czemu tak jest, zostanie wyjaśnione w punkcie zatytułowanym „Uwagi dotyczące struktury leksykalnej”.

Komentarze

W kodzie skryptu można umieszczać komentarze, czyli dowolne uwagi, np. dotyczące jego działania czy określające dane autora. Każdy taki komentarz musi być odpowiednio wyróżniony, aby przeglądarka mogła oddzielić go od formalnych instrukcji. W JavaScriptie mamy do dyspozycji dwa typy komentarzy, oba są zapożyczone z języka C++.

Komentarz wierszowy

Komentarz wierszowy (liniowy) zaczyna się od znaków // i obowiązuje do końca danej linii skryptu. Wszystko, co występuje po tych dwóch znakach, aż do końca bieżącej linii, jest ignorowane przez przeglądarkę przetwarzającą skrypt. Na listingu 2.9 widoczny jest skrypt, w którym umieszczony został komentarz wierszowy.

Listing 2.9. Użycie komentarza wierszowego

```
//Treść warstwy dataDiv zapisz w zmiennej str
var str = "Napis na stronie WWW.";
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Komentarz blokowy

Komentarz blokowy rozpoczyna się od znaków /* i kończy znakami */. Wszystko, co znajduje się pomiędzy, jest pomijane przy przetwarzaniu kodu przez przeglądarkę. Komentarz blokowy można umieścić praktycznie w dowolnym miejscu skryptu, może nawet znaleźć się w środku instrukcji (pod warunkiem, że nie zostanie przedzielone żadne słowo). Przykład użycia komentarza blokowego jest widoczny na listingu 2.10.

Listing 2.10. Użycie komentarza blokowego

```
var str = "Napis na stronie WWW.";
/*
Ta część kodu jest wspólna dla wszystkich skryptów.
Powoduje ona potraktowanie zawartości zmiennej str jako
danych HTML dla warstwy dataDiv.
*/
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Ukrywanie kodu przed przeglądarką

W czasach, gdy JavaScript dopiero zdobywał popularność, istniały przeglądarki nie-rozpoznające znacznika <script>. Jeśli go napotkały, po prostu wyświetlały jego zawartość na stronie. W skryptach osadzonych na witrynie pojawiała się wtedy ich treść. Aby temu zapobiec, stosowano pewną sztuczkę, polegającą na użyciu kombinacji komentarzy JavaScript i HTML. Skrypt należało „opakować” w następujący sposób:

```
<script type="text/javascript">
<!-- Ukrycie przed przeglądarkami nieobsługującymi JavaScriptu
/* ... tutaj treść skryptu ... */
// Koniec kodu JavaScript -->
</script>
```

Za znacznikiem <script> umieszczono symbol początku komentarza HTML <!--. Przeglądarka rozpoznająca znacznik <script> po prostu go pominie. Cały pierwszy wiersz potraktuje więc jak komentarz, natomiast wszystkie kolejne, aż do znacznika

</script>, jako kod skryptu. Ponieważ przed znacznikiem kończącym komentarz HTML umieściliśmy zwykły napis, wiersz ten też musiał być ujęty w komentarz JavaScript:

```
// Koniec kodu JavaScript -->
```

Inaczej napis Koniec kodu JavaScript zostałby potraktowany jak instrukcja.

Przeglądarka nieroznająca znacznika </script> traktuje jego zawartość jak kod HTML. Ona również natrafia na początek komentarza HTML, a zatem wszystko, począwszy od znaków <!-- , a skończywszy na znakach -->, traktuje jak komentarz i nie wyświetla na stronie. A właśnie to było naszym celem.

Obecnie jednak sposób ten stracił na znaczeniu i praktycznie nie jest już stosowany. Wszystkie przywojone przeglądarki prawidłowo rozpoznają znacznik <script> nawet wtedy, jeśli nie obsługują skryptów.

Uwagi dotyczące struktury leksykalnej

Wielkość liter

W JavaScriptie rozróżniane są duże i małe litery. Zatem nie można ich używać zamiennie. Wszystkie instrukcje języka korzystają z małych liter — przekonamy się o tym w kolejnych lekcjach — i nie można tego zmieniać. Należy zwrócić uwagę na nazewnictwo własnych struktur, np. zmiennych. Jak już wspomniano, Liczba i liczba to dwa zupełnie różne identyfikatory i nie można stosować ich zamiennie. Zwracajmy więc uwagę na wielkość liter.

Białe znaki

Wszystkie tzw. białe znaki, tzn. spacje, tabulatory i (ale tylko częściowo, podpunkt „Enter i średnik”) znaki końca wiersza, w trakcie interpretacji kodu przez przeglądarkę są pomijane. To oznacza, że między instrukcje skryptu możemy wstawić dowolną liczbę np. spacji (50, 100 czy nawet 100 000), a i tak zostanie poprawnie zinterpretowany. To sprawia, że możemy tak ułożyć kod skryptu, aby był dla nas jak najbardziej czytelny.

Identyfikatory

Identyfikatory (z ang. *identifiers*) to po prostu nazwy użyte w skrypcie, np. nazwy zmiennych. Przy ich tworzeniu obowiązują następujące zasady.

- ◆ Identyfikator musi zaczynać się od litery, znaku podkreślenia lub znaku dolara.
- ◆ Po pierwszym znaku może wystąpić dowolna liczba liter, cyfr, znaków pokreślenia lub znaków dolara.

Prawidłowymi identyfikatorami są zatem:

- ◆ _abc,
- ◆ _123,
- ◆ zmienna,
- ◆ NUMER.

Znak dolara może być używany od wersji 1.1 JavaScriptu, nie jest to jednak zalecane, gdyż zwykle wykorzystywany jest przez narzędzia do automatycznej generacji kodu.

Polskie litery i znaki niestandardowe

W JavaScriptie można używać znaków spoza standardowego zestawu ASCII. We wcześniejszych wersjach możliwe to było tylko w łańcuchach znakowych (fragmentach kodu ujętych w znaki cudzysłowu lub apostrofu) oraz komentarzach. Standard ECMAScript v3 wprowadził pełną obsługę Unicode. Oznacza to, że we wszelkich identyfikatorach można używać znaków pochodzących z dowolnego języka świata. Przykładowo nazwy zmiennych z powodzeniem mogą zawierać polskie znaki diakrytyczne (a także znaki niemieckie, francuskie, chińskie i inne). Najczęściej jednak nie korzysta się z tego udogodnienia i używa tylko podstawowego alfabetu łacińskiego (często też stosuje się nazwy identyfikatorów, które pochodzą z języka angielskiego). W książce będą używane identyfikatory zawierające polskie litery.

Enter i średnik

Wcześniej padło już stwierdzenie, że każda instrukcja powinna być zakończona średnikiem. Jest prawdziwe i dokładnie tak powinniśmy postępować. Trzeba jednak wiecieć, że formalnie nie jest to obligatoryjne, bowiem instrukcję można też zakończyć znakiem końca wiersza (czyli, mówiąc popularnie, wciskając *Enter*). I tak skrypt z listingu 2.4 zapisany w następujący sposób:

```
var str = "Umiem wyświetlać napisy!"  
var dataDiv = document.getElementById("dataDiv")  
dataDiv.innerHTML = str
```

będzie formalnie poprawny. Przeglądarka sama „dostawi” brakujące średniki na końcu każdego wiersza. Zdecydowanie należy jednak unikać takiego postępowania, gdyż może prowadzić do powstawania błędów i zmniejsza czytelność kodu.

Słowa zarezerwowane

W JavaScriptie, jak w każdym języku programowania, istnieje zestaw słów zarezerwowanych. Oznaczają one różne konstrukcje języka i nie wolno ich używać jako identyfikatorów (np. jako nazw zmiennych). Słów zarezerwowanych stosowane jako część języka według ECMA v3 zostały zebrane w tabeli 2.2. Natomiast słowa niestanowiące konstrukcji języka według ECMA v3, ale zarezerwowane dla przyszłych zastosowań wyliczono w tabeli 2.3.

Tabela 2.2. Słowa zarezerwowane według ECMA v3

break	default	false	if	null	throw	var
case	delete	finally	in	return	true	void
catch	do	for	instanceof	switch	try	while
continue	else	function	new	this	typeof	with

Tabela 2.3. Słowa zarezerwowane do przyszłych zastosowań

abstract	const	extends	import	package	static	volatile
boolean	debugger	final	int	private	super	
byte	double	float	interface	protected	synchronized	
char	enum	goto	long	public	throws	
class	export	implements	native	short	transient	

Instrukcja document.write

W JavaScriptie dostępna jest instrukcja `document.write`, która pozwala umieścić ciąg znaków bezpośrednio w kodzie strony HTML. Taki ciąg znaków zostanie umieszczony w treści strony dokładnie w miejscu zapisania tej instrukcji. Niegdyś była bardzo popularna, ale działa tylko w kodzie HTML (nie działa w kodzie XHTML). Dlatego też nie będziemy jej stosować w dalszej części książki. Warto jednak wiedzieć, że taka konstrukcja istnieje. Należy znać sposób jej użycia, bo wiele dostępnych w internecie skryptów wciąż z niej korzysta.

Ogólna postać instrukcji `document.write` jest następująca:

```
document.write("ciąg znaków");
```

Jeśli w kodzie strony ma się pojawić napis: Umiem korzystać z instrukcji `document.write`, należy użyć polecenia:

```
document.write("Umiem korzystać z instrukcji document.write.");
```

Skrypt zawierający to polecenie powinien zostać umieszczony w tym miejscu strony, w którym ma się pojawić napis, np. wewnątrz akapitu tekstowego zdefiniowanego za pomocą znacznika `<p>`. Taki przykład umieszczono na listingu 2.11.

Listing 2.11. Użycie instrukcji `document.write`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
</head>
```

```
<body>
<p>
<script type="text/javascript">
    document.write("Umiem korzystać z instrukcji document.write.");
</script>
</p>
</body>
</html>
```

Układ strony jest typowy dla HTML. Jedynie zawartość znacznika <p> jest generowana przez skrypt. Znaczenie instrukcji document.write to nic innego jak polecenie: „Umieść w dokumencie ciąg znaków znajdujący się między znakami cudzysłowu”. A zatem przedstawiony kod zachowałby się tak, jakby znacznik <p> miał postać:

```
<p>
Umiem korzystać z instrukcji document.write.
</p>
```

Na stronie pojawi się więc żądanego napis.

Takie postępowanie było bardzo wygodne dla twórców skryptów, jednak wraz ze zwiększającą się popularnością XHTML niezbędne stało się stosowanie innych metod. Więcej o współpracy skryptów z przeglądarką i dokumentami HTML (XHTML) dowiemy się w rozdziale 4.

Lekcja 3. Operacje i operatory

Lekcja 3. poświecona jest operatorom i operacjom, jakie można za ich pomocą wykonywać. Poznamy najprostsze i doskonale wszystkim znane operacje arytmetyczne, takie jak dodawanie i mnożenie, ale też niespotykane w życiu, poza programistycznym, operacje inkrementacji i dekrementacji, a także złożone operacje przypisania. Dowiemy się, jakie priorytety mają poszczególne operatory. Nieco miejsca poświęcimy także operacjom wykonywanym na ciągach znaków.

Wykonywanie operacji

W JavaScriptie, podobnie jak w innych językach programowania, występują operatory pozwalające na wykonywanie rozmaitych operacji. Operatory możemy podzielić na następujące grupy:

- ◆ arytmetyczne,
- ◆ porównywania (relacyjne),
- ◆ bitowe,
- ◆ logiczne,
- ◆ przypisania,
- ◆ pozostałe.

Mogliśmy więc wykonywać operacje dodawania, odejmowania, porównywania, przypisania i wiele innych. Operacje wykonywane są na operandach, czyli argumentach operatorów. Jeśli wykonujemy operację dodawania, np.:

$2 + 5$

argumentami operatora dodawania + są dwa operandy, czyli liczby 2 i 5.

W książce został już użyty jeden z operatorów. Był to operator przypisania =. Powoduje on przypisanie operandowi znajdującemu się z lewej strony tego, co mamy po prawej. Kiedy więc pisaliśmy:

```
var str = "abc";
```

dokonywaliśmy przypisania ciągu znaków abc zmiennej o nazwie str.

Za pomocą operatorów możemy budować bardziej złożone wyrażenia. Można wykonać ciąg dodawań, np.:

```
var suma = 1 + 3 + 5;
```

Takie wyrażenie będzie przetwarzane od strony prawej do lewej (jest to ogólna zasada dotycząca przetwarzania wyrażeń), czyli najpierw zostanie wykonane działanie $5 + 3$, a w jego miejsce zostanie podstawiona wynikająca z tej operacji wartość, czyli 8. Powstanie więc wyrażenie:

```
var suma = 1 + 8;
```

Następnie zostanie wykonane działanie $8 + 1$, a w jego miejsce zostanie podstawiony wynik tej operacji, czyli 9. Powstanie wtedy wyrażenie:

```
var suma = 9;
```

Zostanie ono wykonane już bezpośrednio, czyli ostatecznie zmiennej suma zostanie przypisana wartość 9. Warto też wiedzieć, że wynikiem operacji przypisania (tak, operacja przypisania ma swój wynik) jest wartość, która została przypisana. A zatem w miejscu wystąpienia instrukcji:

```
liczba = 5;
```

znajdzie się wartość 5.

Stosując opisane zasady, można napisać przykładowe (w pełni poprawne i przydatne) wyrażenie:

```
var wartosc = suma = 4 + 8;
```

Po powyższych wyjaśnieniach z pewnością stało się jasne, że najpierw zostanie wykonane dodawanie, jego wynik zostanie przypisany zmiennej suma (jeżeli nie została zadeklarowana wcześniej, zostanie w tym miejscu utworzona), wynik tego przypisania, czyli wartość zmiennej suma, zostanie przypisany zmiennej wartosc.

O wyświetlaniu danych raz jeszcze

Skrypty z tego rozdziału, co zostało zaznaczone w lekcji 2., wymagają, aby wszystkie dane, które mają się pojawić na stronie WWW, zostały zapisane w zmiennej o nazwie str. Kiedy danych jest niewiele, nie stanowi to żadnego problemu. Jeśli chcemy wyświetlić ciąg znaków, piszemy np.:

```
str = "abc";
```

A kiedy mamy zamiar wyświetlić liczbę, piszemy:

```
str = 54;
```

Co jednak powinniśmy zrobić, gdy dane do wyświetlenia będą tworzone w trakcie działania skryptu lub też składane z mniejszych porcji? Rozważmy sytuację, gdy na stronie ma się pojawić ciąg liter abcdef, ale z pewnych względów musi być złożony z dwóch części: pierwszej — abc i drugiej — def. Jeśli użyjemy instrukcji:

```
str = "abc";
str = "def";
```

nie osiągniemy założonego celu. Pierwszy wiersz powoduje przypisanie zmiennej str ciągu abc i po jego wykonaniu będzie zawierała ten ciąg. Jednak drugi wiersz to przyisanie zmiennej str ciągu def, tym samym wcześniejsza jej zawartość zostanie skasowana i zmienna będzie teraz zawierała napis def. Tymczasem zadanie trzeba wykonać tak, aby dotycząca wartość zmiennej nie została utracona. Jak to zrobić? Należy skorzystać z zasad przedstawionych w poprzedniej części lekcji. Prawidłowy ciąg instrukcji będzie zatem miał postać:

```
str = "abc";
str = str + "def";
```

Najpierw zmiennej str zostanie przypisany ciąg znaków abc. Pierwsza instrukcja ma bowiem taką samą postać jak w poprzednim przykładzie. Druga instrukcja jest jednak inna. Oznacza ona: „Dodaj do aktualnej wartości zapisanej w str ciąg def i przypisz wynik tej operacji zmiennej str”, czyli najpierw zostanie wykonane działanie str $\rightarrow+$ "def". Ponieważ w str znajduje się ciąg abc, wynikiem tej operacji będzie ciąg abcdef. Wynik zostanie podstawiony z prawej strony operatora przypisania =. A zatem powstanie działanie str = abcdef, które już bezpośrednio przypisze wspomniany ciąg zmiennej str. Jak to wyglądałoby w praktyce, zobrazowano w kodzie z listingu 2.12.

Listing 2.12. Wyświetlanie danych złożonych z mniejszych fragmentów

```
var str = "To ";
str = str + "jest przykład ";
str = str + "uzupełniania ";
str = str + "wartości zmiennej.";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Po wykonaniu pierwszych czterech instrukcji zmienna str będzie zawierała ciąg znaków To jest przykład uzupełniania wartości zmiennej. i po wczytaniu skryptu do przeglądarki taki napis pojawi się na stronie.

Operatory arytmetyczne

Operatory arytmetyczne służą do wykonywania operacji arytmetycznych. To dobrze wszystkim znane dodawanie, odejmowanie, mnożenie i dzielenie. Występuje tu także tzw. dzielenie modulo, czyli reszta z dzielenia. Do tej grupy zaliczmy również operatory ustalające znak wartości, czyli to, czy jest dodatnia, czy ujemna. Operatory arytmetyczne zostały zebrane w tabeli 2.4.

Tabela 2.4. Operatory arytmetyczne

Operator	Wykonywane działanie	Przykład
*	mnożenie	$x * y$
/	dzielenie	x / y
+	dodawanie	$x + y$
-	odejmowanie	$x - y$
%	dzielenie modulo (reszta z dzielenia)	$x \% y$
+	ustalenie znaku wartości	+x
-	ustalenie znaku wartości	-x

Przy wykonywaniu operacji arytmetycznych obowiązują standardowe reguły matematyki. Podczas zapisywania wyrażeń obliczeniowych można stosować nawiasy zmieniające kolejność działań lub zwiększające czytelność zapisu. Kilka prostych operacji matematycznych wykonuje skrypt z listingu 2.13.

Listing 2.13. Proste operacje matematyczne

```

var suma = 5 + 2;
var różnica = 8 - 3;
var iloczyn = 2 * 3;
var iloraz = 10 / 2;
var wynik = 2 * (12 - 5) + 15 / (9 - 6);
var str = "suma (5 + 2) = " + suma + "<br />";
str = str + "różnica (8 - 3) = " + różnica + "<br />";
str = str + "iloczyn (2 * 3) = " + iloczyn + "<br />";
str = str + "iloraz (10 / 2) = " + iloraz + "<br />";
str = str + "wynik (2 * (12 - 5) + 15 / (9 - 6)) = ";
str = str + wynik + "<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Zostało tu zadeklarowanych 5 zmiennych, suma, różnica, iloczyn, iloraz oraz wynik, i każdej z nich został przypisany wynik pewnej prostej operacji arytmetycznej. Następnie wartość każdej z tych zmiennych została połączona z opisującym ją ciągiem znaków i przypisana zmiennej str, co sprawiło, że pojawiła się na ekranie. Przykładowo operacja:

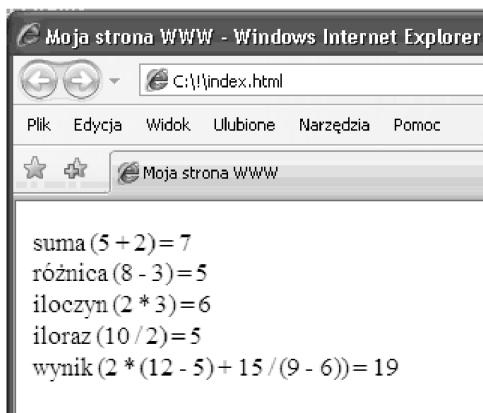
```
var str = "suma (5 + 2) = " + suma + "<br />";
```

oznacza: „Pomiędzy ciągi znaków suma (5 + 2) = i
 wstaw wartość zmiennej suma, a wynik tej operacji przypisz zmiennej str”.

Na końcu każdego tworzonego wiersza umieszczany jest znacznik
, aby wyniki na ekranie pojawiły się jeden pod drugim, a nie w jednej linii. Wynik działania skryptu został przedstawiony na rysunku 2.4.

Rysunek 2.4.

Wynik działania skryptu 2.13



Nic też nie stoi na przeszkodzie, aby wykonywać operacje arytmetyczne na zmiennych czy też liczbach i zmiennych. Można więc np. pomnożyć dwie zmienne (aściślej, wartości w nich zawarte) lub też dodać liczbę do zmiennej. Prawidłowe są zatem następujące operacje:

```
var wartosc = 10;
var wynik = wartosc * 15;
```

I poniższe:

```
var wartosc1 = 8;
var wartosc2 = 4;
var wynik = wartosc1 + wartosc2;
```

Mögemy również wykonywać operacje na liczbach z częścią ułamkową, ale wtedy jako separatora dziesiętnego używamy znaku kropki (a nie — zgodnie z polską notacją — przecinka) np.:

```
var wartosc = 1.25;
var wynik = wartosc * 8.34;
```

Przyjrzyjmy się teraz operatorowi dzielenia modulo %. Podaje on po prostu resztę z dzielenia dwóch wartości. Przykładowo działanie $10 \% 3$ da w wyniku 1. Trójka zmieści się bowiem w dziesięciu 3 razy, pozostawiając resztę 1 ($3 * 3 = 9$, $9 + 1 = 10$). Podobnie $21 \% 8 = 5$, ponieważ $2 * 8 = 16$, $16 + 5 = 21$. W wielu przypadkach możliwość wykonywania takiej operacji będzie bardzo pomocna.

Pozostały jeszcze operatory ustalenia znaku wartości, czyli jednoargumentowe + i -. Jak działają? Dokładnie tak, jak uczyono nas na lekcjach matematyki. Wynika z tego, że praktyczne znaczenie ma jedynie operator - (+ bowiem w żadnym wypadku nie zmienia znaku wartości, jednak formalnie operator ten występuje i może zostać użyty). Spójrzmy na kilka przykładów:

```
var wartosc1 = +5;
var wartosc2 = -5;
var wartosc3 = +wartosc1;
var wartosc4 = -wartosc1;
var wartosc5 = -wartosc2;
```

Wydają się dosyć oczywiste. Zmienna wartosc1 otrzymuje wartość 5, zmienna wartosc2 — -5. W związku z tym, zmienna wartosc3 będzie miała wartość 5, zmienna wartosc4 — -5 (nastąpiła zmiana znaku wartości zmiennej wartosc1), a wartosc5 — 5 (nastąpiła zmiana znaku wartości zmiennej wartosc2).

Operatorы inkrementacji i dekrementacji

Inkrementacja to po prostu zwiększanie, a dekrementacja — zmniejszanie. Operatory te zaliczamy do arytmetycznych, jednak zostaną omówione osobno, ze względu na to, że nie są powszechnie znane (choćby z lekcji matematyki).

Operator inkrementacji powoduje przyrost wartości zmiennej o jeden. Operator ten, zapisywany jako „++”, może występować w dwóch formach: przyrostkowej bądź przedrostkowej. Oznacza to, że jeśli mamy zmienną, która nazywa się np. x, forma przedrostkowa będzie wyglądać tak: `++x`, natomiast przyrostkowa tak: `x++`. Oba te wyrażenia zwiększą wartość zmiennej x o jeden, jednak wcale nie są równoważne. Otóż, operacja `x++` zwiększa wartość zmiennej po jej wykorzystaniu, natomiast `++x` — przed wykorzystaniem. Takie rozróżnienie, choć osobom poczatkującym zapewne wydaje się niebyt zrozumiałe, może być bardzo pomocne podczas pisania skryptów. Różnice w działaniu najlepiej sprawdzić na konkretnym przykładzie. Widnieje on na listingu 2.14.

Listing 2.14. Działanie operatora inkrementacji

```
var x = 10;
var y;
var str = "Uzyskane wartości: ";

/*1*/ str = str + x++;
/*2*/ str = str + " ";
/*3*/ str = str + ++x;
/*4*/ str = str + " ";
/*5*/ str = str + x;
```

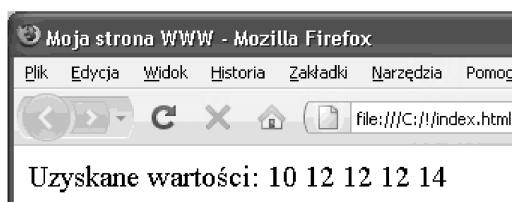
```
/*6*/ str = str + " ";
/*7*/ y = x++;
/*8*/ str = str + y;
/*9*/ str = str + " ";
/*10*/ y = ++x;
/*11*/ str = str + y;

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Wynikiem działania skryptu (dla ułatwienia opisu ważniejsze wiersze zostały ponumerowane) będzie ciąg znaków 10 12 12 12 14, widoczny na rysunku 2.5. Dlaczego? Otóż, w wierszu oznaczonym numerem 1 najpierw aktualna wartość zmiennej x ($x = 10$) jest dodawana do ciągu str, a następnie zmieniona jest zwiększana o 1 (czyli $x = 11$). W linii 3. najpierw wartość x jest zwiększana o 1 ($x = 12$), a następnie dodawana do ciągu str. W wierszu 5. aktualna wartość zmiennej x , czyli 12, jest dodawana do str. W wierszu 7. zmiennej y jest przypisywana wartość zmiennej x ($x = 12$), a następnie zmieniona x jest zwiększana o 1 (czyli $y = 12$, $x = 13$). W wierszu 8. aktualna wartość zmiennej y , czyli 12, jest dodawana do str. W wierszu 10. najpierw jest zwiększana wartość zmiennej x o 1 (czyli $x = 14$), a następnie wartość ta jest przypisywana zmiennej y (czyli $y = 14$ i $x = 14$). Na początku może wydawać się to nieco skomplikowane, ale po dokładnym przeanalizowaniu i samodzielnym wykonaniu kilku własnych ćwiczeń operator ten nie powinien sprawiać żadnych kłopotów.

Rysunek 2.5.

Wynik działania skryptu ilustrującego działanie operatora `++`



Zwróćmy jeszcze uwagę na zapis w wierszu 3. Ma on postać:

```
str = str + ++x;
```

Występują tu obok siebie trzy znaki `+`, co może wyglądać nieco dziwnie, trzeba sobie jednak uświadomić, że są to po prostu dwa operatory. Pierwszy to operator dodawania `+`, a drugi operator inkrementacji przedrostkowej `++`. Między tymi operatorami musi wystąpić odstęp, inaczej przeglądarka nie byłaby w stanie zinterpretować takiego wyrażenia.

Operator dekrementacji działa analogicznie do operatora inkrementacji, z tym że zamiast zwiększać wartości zmiennych, zmniejsza je o jeden. Zmieńmy zatem kod z listingu 2.14 tak, aby wszystkie wystąpienia operatora `++` zostały zamienione na `--`. Otrzymamy wtedy skrypt widoczny na listingu 2.15.

Listing 2.15. Użycie operatora dekrementacji

```

var x = 10;
var y;
var str = "Uzyskane wartości: ";

/*1*/ str = str + x--;
/*2*/ str = str + " ";
/*3*/ str = str + --x;
/*4*/ str = str + " ";
/*5*/ str = str + x;
/*6*/ str = str + " ";
/*7*/ y = x--;
/*8*/ str = str + y;
/*9*/ str = str + " ";
/*10*/ y = --x;
/*11*/ str = str + y;

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Wynikiem działania tego skryptu będzie ciąg znaków 10 8 8 8 6. W wierszu oznaczonym numerem 1 najpierw aktualna wartość zmiennej x ($x = 10$) jest dodawana do ciągu str, a następnie zmieniona jest zmniejszana o 1 (czyli $x = 9$). W linii 3. najpierw wartość x jest zmniejszana o 1 ($x = 8$), a następnie dodawana do ciągu str. W wierszu 5. aktualna wartość zmiennej x , czyli 8, jest dodawana do str. W wierszu 7. zmiennej y jest przypisywana wartość zmiennej x ($x = 8$), a następnie zmieniona x jest zmniejszana o 1 (czyli $y = 8$, $x = 7$). W wierszu 8. aktualna wartość zmiennej y , czyli 8, jest dodawana do str. W wierszu 10. najpierw jest zmniejszana wartość zmiennej x o 1 (czyli $x = 6$), a następnie wartość ta jest przypisywana zmiennej y (czyli $y = 6$ i $x = 6$).

Uwaga: operatory inkrementacji i dekrementacji mogą być używane tylko w stosunku do zmiennych, elementów tablic (lekция 7.) i właściwości obiektów (lekция 9.). Nie można ich używać np. do wartości bezpośrednio umieszczonych w kodzie. Przykładowo konstrukcja:

`25++;`

jest błędna.

Trzeba też wiedzieć, że wynikiem działania operatorów `++` lub `--` jest zawsze wartość, zatem w miejscu wystąpienia przykładowej instrukcji:

`liczba++;`

zostanie podstawiona wartość zmiennej `liczba`. Dlatego też nie można używać konstrukcji w postaci:

`--liczba++;`

czy też

`(liczba++)--;`

Operatory porównywania (relacyjne)

Operatory porównywania, czyli relacyjne, służą do porównywania argumentów². Wynikiem takiego porównania (wyrażenia zawierającego operator relacyjny) jest wartość logiczna true (jeśli jest ono prawdziwe) lub false (jeśli jest fałszywe). Zatem wynikiem operacji `argument1 == argument2` będzie true, jeżeli argumenty są sobie równe, oraz false, jeżeli argumenty są różne. Czyli `4 == 5` ma wartość false, `2 == 2` ma wartość true, a `8 < 3` — false. Do dyspozycji mamy operatory porównania zawarte w tabeli 2.5. Z operatorów relacyjnych będziemy korzystać już w lekcji 3., omawiającej instrukcje warunkowe. Tam też zostaną przedstawione konkretne przykłady ich wykorzystania.

Tabela 2.5. Operatory porównywania

Operator	Opis	Przykład
<code>==</code>	Wynikiem jest true, jeśli argumenty są sobie równe. W przeciwnym przypadku wynikiem jest false.	<code>x == y</code>
<code>!=</code>	Wynikiem jest true, jeśli argumenty są różne. W przeciwnym przypadku wynikiem jest false.	<code>x != y</code>
<code>==</code>	Wynikiem jest true, jeśli oba argumenty są tego samego typu i są sobie równe. W przeciwnym przypadku wynikiem jest false.	<code>x === y</code>
<code>!==</code>	Wynikiem jest true, jeśli argumenty są różne bądź są różnych typów. W przeciwnym przypadku wynikiem jest false.	<code>x !== y</code>
<code>></code>	Wynikiem jest true, jeśli argument lewostronny jest większy od prawostronnego. W przeciwnym przypadku wynikiem jest false.	<code>x > y</code>
<code><</code>	Wynikiem jest true, jeśli argument lewostronny jest mniejszy od prawostronnego. W przeciwnym przypadku wynikiem jest false.	<code>x < y</code>
<code>>=</code>	Wynikiem jest true, jeśli argument lewostronny jest większy od prawostronnego lub mu równy. W przeciwnym przypadku wynikiem jest false.	<code>x >= y</code>
<code><=</code>	Wynikiem jest true, jeśli argument lewostronny jest mniejszy od prawostronnego lub mu równy. W przeciwnym przypadku wynikiem jest false.	<code>x <= y</code>

Operatory logiczne

Operacje logiczne mogą być wykonywane na argumentach, które posiadają wartość logiczną prawda (true) lub fałsz (false). Operatory logiczne zostały przedstawione w tabeli 2.6. Operatory `&&` i `||` są dwuargumentowe, natomiast operator `!` jest jednoargumentowy.

² Formalnie dokonuje się rozróżnienia na operatory relacyjne (mniejszy, większy, mniejszy lub równy i większy lub równy) oraz równości (równy, nierówny, identyczny, nieidentyczny).

Tabela 2.6. Operatory logiczne

Operator	Wykonywane działanie	Przykład
&&	iloczyn logiczny (AND)	a && b
	suma logiczna (OR)	a b
!	negacja logiczna (NOT)	!a

Iloczyn logiczny

Wynikiem iloczynu logicznego jest wartość true wtedy i tylko wtedy, kiedy oba argumenty mają wartość true. W każdym innym przypadku wynikiem jest false. Zobrazowano to w tabeli 2.7.

Tabela 2.7. Działanie iloczynu logicznego

Argument 1	Argument 2	Wynik
true	true	true
true	false	false
false	true	false
false	false	false

Suma logiczna

Wynikiem sumy logicznej jest wartość false wtedy i tylko wtedy, kiedy oba argumenty mają wartość false. W każdym innym przypadku wynikiem jest true. Zobrazowano to w tabeli 2.8.

Tabela 2.8. Działanie sumy logicznej

Argument 1	Argument 2	Wynik
true	true	true
true	false	true
false	true	true
false	false	false

Negacja logiczna

Operacja logicznej negacji zamienia wartość argumentu na przeciwną. A zatem, jeśli argument miał wartość true, będzie miał wartość false, i odwrotnie, jeśli miał wartość false, będzie miał wartość true. Spójrzmy na tabelę 2.9.

Tabela 2.9. Działanie negacji logicznej

Argument	Wynik
true	false
false	true

Operatory bitowe

Operatory bitowe pozwalają na wykonywanie operacji na poszczególnych bitach liczb. Aby się z nimi bliżej zapoznać, musimy przypomnieć sobie przynajmniej podstawowe informacje na temat systemu dwójkowego. W systemie dziesiętnym, którego używamy na co dzień, wykorzystywanych jest dziesięć cyfr, od 0 do 9. W systemie dwójkowym będą zatem stosowane jedynie dwie cyfry: 0 i 1. Kolejne liczby są budowane z tych dwóch cyfr, dokładnie tak samo jak w systemie dziesiętnym; przedstawiono to w tabeli 2.10. Widać wyraźnie, że np. 4 dziesiętnie to 100 dwójkowo, a 10 dziesiętnie to 1010 dwójkowo.

Tabela 2.10. Kolejne 15 liczb w systemie dwójkowym i ich odpowiedniki w systemie dziesiętnym

System dwójkowy	System dziesiętny
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Występujące w JavaScript operatory bitowe zostały przedstawione w tabeli 2.11. Są to: iloczyn bitowy (koniuunkcja bitowa, operacja AND), suma bitowa (alternatywa bitowa, operacja OR), negacja bitowa (uzupełnienie do jedynki, operacja NOT), suma bitowa modulo 2 (alternatywa bitowa wykluczająca, różnica symetryczna, operacja XOR) oraz operacje przesunięć bitów. Wszystkie operatory są dwuargumentowe, oprócz operatora bitowej negacji, który jest jednoargumentowy.

Iloczyn bitowy

Iloczyn bitowy to operacja powodująca, że włączone pozostają tylko te bity, które były włączone w obu argumentach. Wynik operacji AND na pojedynczych bitach zobrazowano w tabeli 2.12. Jeśli zatem wykonamy operację:

```
var liczba = 179 & 38;
```

to zmiennej liczba zostanie przypisana wartość 34.

Tabela 2.11. Operatory bitowe

Operator	Wykonywane działanie	Przykład
&	iloczyn bitowy AND	a & b
	suma bitowa OR	a b
~	negacja bitowa NOT	~a
^	bitowa różnica symetryczna XOR	a ^ b
>>	przesunięcie bitowe w prawo	a >> n
<<	przesunięcie bitowe w lewo	a << n
>>>	przesunięcie bitowe w prawo z wypełnieniem zerami	a >>> n

Tabela 2.12. Wyniki operacji AND dla pojedynczych bitów

Argument 1	Argument 2	Wynik
1	1	1
1	0	0
0	1	0
0	0	0

Dlaczego 34? Najłatwiej pokazać to, jeśli obie wartości (czyli 179 i 38) przedstawi się w postaci dwójkowej. 179 w postaci dwójkowej to 10110011, natomiast 38 to 00100110. Operacja AND będzie zatem miała postać:

$$\begin{array}{r}
 10110011 \text{ (179)} \\
 00100110 \text{ (38)} \\
 \hline
 00100010 \text{ (34)}
 \end{array}$$

Wynikiem jest więc 34.

Suma bitowa

Suma bitowa to operacja powodująca, że pozostają włączone te bity, które były włączone przynajmniej w jednym z argumentów. Wynik operacji OR na pojedynczych bitach można zobaczyć w tabeli 2.13. Jeśli zatem wykonamy operację:

`var liczba = 34 | 65;`

to zmiennej `liczba` zostanie przypisana wartość 99.

Tabela 2.13. Wyniki operacji OR dla pojedynczych bitów

Argument 1	Argument 2	Wynik
1	1	1
1	0	1
0	1	1
0	0	0

Jeśli obie liczby rozpiszemy w postaci dwójkowej, otrzymamy 00100010 (34) i 01000001 (65). Zatem całe działanie będzie miało postać:

$$\begin{array}{r} 00100010 \text{ (34)} \\ 01000001 \text{ (65)} \\ \hline 01100011 \text{ (99)} \end{array}$$

Negacja bitowa

Negacja bitowa powoduje zmianę stanu bitów. A zatem tam, gdzie dany bit miał wartość 0, będzie miał 1, natomiast tam, gdzie miał wartość 1, będzie miał 0. Działanie operacji NOT na pojedynczych bitach zobrazowano w tabeli 2.14.

Tabela 2.14. Wyniki operacji NOT dla pojedynczych bitów

Argument	Wynik
1	0
0	1

Bitowa różnica symetryczna

Bitowa różnica symetryczna, czyli operacja XOR, powoduje, że włączone zostają te bity, które miały różne stany w obu argumentach, a pozostałe zostają wyłączone. Wynik operacji XOR na pojedynczych bitach można zobaczyć w tabeli 2.15.

Tabela 2.15. Wyniki operacji XOR dla pojedynczych bitów

Argument 1	Argument 2	Wynik
1	1	0
1	0	1
0	1	1
0	0	0

Wykonanie przykładowej operacji:

```
var liczba = 34 ^ 118;
```

spowoduje przypisanie zmiennej liczba wartości 84. Jeśli bowiem zapiszemy obie wartości w postaci dwójkowej, to 34 przyjmie postać 00100010, natomiast 118 — 01110110. Operacja XOR będzie zatem wyglądała następująco:

$$\begin{array}{r} 00100010 \text{ (34)} \\ 01110110 \text{ (118)} \\ \hline 01010100 \text{ (84)} \end{array}$$

Przesunięcie bitowe w lewo

Przesunięcie bitowe w lewo to operacja polegająca na przesunięciu wszystkich bitów argumentu znajdującego się z lewej strony operatora w lewo o liczbę miejsc wskazaną przez argument znajdujący się z jego prawej strony. Wykonanie przykładowej operacji:

```
var liczba = 84 << 1;
```

spowoduje przypisanie zmiennej `liczba` wartości 168. Działanie `84 << 1` oznacza bowiem: „Przesuń wszystkie bity wartości 84 o jedno miejsce w prawo”. Skoro 84 w postaci dwójkowej ma postać 01010100, to po przesunięciu powstanie 10101000, czyli 168.

Warto zauważyć, że przesunięcie bitowe w lewo odpowiada mnożeniu wartości przez wielokrotność liczby 2. I tak przesunięcie w lewo o jedno miejsce to pomnożenie przez 2, o dwa miejsca — przez 4, o trzy miejsca — przez 8 itd.

Przesunięcie bitowe w prawo

Analogicznie do powyższego, przesunięcie bitowe w prawo polega na przesunięciu wszystkich bitów argumentu znajdującego się z lewej strony operatora w prawo o liczbę miejsc wskazaną przez argument, który znajduje się z prawej strony operatora. A zatem wykonanie operacji:

```
var liczba = 84 >> 1;
```

spowoduje przypisanie zmiennej `liczba` wartości 42. Oznacza to bowiem przesunięcie wszystkich bitów wartości 01010100 o jedno miejsce w prawo, czyli powstanie wartości 00101010 dwójkowo (42 dziesiętnie).

Tu również należy zwrócić uwagę, że przesunięcie bitowe w prawo odpowiada podzieleniu wartości przez wielokrotność liczby 2, czyli przesunięcie w prawo o jedno miejsce to podzielenie przez 2, o dwa miejsca — przez 4, o trzy miejsca — przez 8 itd.³

Operatory przypisania

Operatory przypisania są dwuargumentowe i powodują przypisanie wartości argumentu znajdującego się z prawej strony operatora temu, który znajduje się z lewej. Taką najprostszą operację już poznaliśmy, odbywa się przy wykorzystaniu operatora `=` (równa się). Jeśli napiszemy `liczba = 10`, oznacza to, że zmiennej `liczba` chcemy przypisać wartość 10.

W JavaScriptie istnieje jednak również cały zestaw operatorów łączących operacje przypisania z inną — arytmetyczną lub bitową. Przykładowo mamy operator `+=`, który oznacza: „Przypisz argumentowi umieszczonemu z lewej strony wartość wynikającą z dodawania argumentów znajdujących się z lewej i prawej strony operatora”.

³ Należy jednak pamiętać, że jeżeli dzielona liczba będzie nieparzysta, w wyniku takiego dzielenia zostanie utracona część ułamkowa.

Choć brzmi to z początku nieco zawile, w rzeczywistości jest bardzo proste i znacznie upraszcza niektóre konstrukcje programistyczne. Po prostu przykładowy zapis:

`liczba += 5`

tłumaczymy jako:

`liczba = liczba + 5`

Oznacza on: „Przypisz zmiennej `liczba` wartość wynikającą z dodawania `liczba + 5`” lub — jeszcze prościej — zwięksź wartość zmiennej `liczba` o 5.

Operatory tego typu występujące w JavaScriptie zostały zebrane w tabeli 2.16. Schematycznie możemy przedstawić ich znaczenie następująco:

`arg1 op= arg2`

oznacza działanie

`arg1 = arg1 op arg2`

czyli `a += b` oznacza `a = a + b`, `a *= b` oznacza `a = a * b`, `a %= b` oznacza `a = a % b` itd.

Tabela 2.16. Operatory przypisania i ich znaczenie

Argument 1	Operator	Argument 2	Znaczenie
x	=	y	<code>x = y</code>
x	<code>+=</code>	y	<code>x = x + y</code>
x	<code>-=</code>	y	<code>x = x - y</code>
x	<code>*=</code>	y	<code>x = x * y</code>
x	<code>/=</code>	y	<code>x = x / y</code>
x	<code>%=</code>	y	<code>x = x % y</code>
x	<code><<=</code>	y	<code>x = x << y</code>
x	<code>>>=</code>	y	<code>x = x >> y</code>
x	<code>>>>=</code>	y	<code>x = x >>> y</code>
x	<code>&=</code>	y	<code>x = x & y</code>
x	<code> =</code>	y	<code>x = x y</code>
x	<code>^=</code>	y	<code>x = x ^ y</code>

Od tej pory w celu dodania wartości do zmiennej str (a tym samym, wyświetlenia tej wartości na ekranie) nie będziemy już musieli stosować konstrukcji:

`str = str + wartość;`

Zamiast tego będziemy używali operatora `+=`:

`str += wartość;`

co będzie miało takie samo znaczenie, a znacznie uprości zapis.

Operatory przypisania mają prawe wiązanie, co oznacza, wyrażenie będzie przetwarzane od strony prawej do lewej. Wynikiem operacji przypisania jest natomiast przypisana wartość. Dzięki temu można zastosować instrukcję typu:

```
x = y = z = 15;
```

Oznacza ona przypisanie wszystkim zmiennym — x , y i z — wartości 15. Najpierw bowiem zostanie wykonana instrukcja $z = 15$, a w jej miejsce wstawiona wartość 15. Wyrażenie uprości się zatem do postaci:

```
x = y = 15;
```

Następnie zostanie wykonana instrukcja $y = 15$ itd.

Operator warunkowy

Operator warunkowy zostanie omówiony w lekcji 4. dotyczącej instrukcji warunkowych.

Operator typeof

Operator `typeof` pozwala na uzyskanie informacji o typie danej wartości, najczęściej zmiennej. Używamy go w sposób następujący:

```
typeof wartość
```

Wynikiem działania operatora jest ciąg znaków określający typ wartości. Może to być ciąg:

- ◆ `string` — gdy wartość jest ciągiem znaków;
- ◆ `number` — gdy wartość jest numeryczna;
- ◆ `boolean` — gdy wartość jest typu boolowskiego;
- ◆ `object` — gdy wartość jest obiektem bądź tablicą⁴;
- ◆ `function` — gdy wartość jest funkcją.

Przykładowo zapis:

```
typeof "abc"
```

da w wyniku ciąg `string`, a:

```
typeof 2.14
```

da w wyniku ciąg `number`.

Konkretny przykład zastosowania operatora `typeof` umieszczono w dalszej części książki.

⁴ Uwaga! Zapis `typeof null` da w wyniku również ciąg `object`.

Pozostałe operatory

W JavaScriptie występuje jeszcze kilka innych operatorów, których jednak nie będziemy dokładnie omawiać. Są to m.in. operatory indeksowania tablic, wywołania funkcji, rozdzielania wyrażeń, tworzenia obiektów itp. Pojawią się one w dalszej części książki w trakcie omawiania kolejnych tematów, zostały też uwzględnione w tabeli prezentującej priorytety operatorów.

Priorytety operatorów

Oprócz znajomości operatorów, niezbędna jest jeszcze wiedza na temat ich priorytetów, czyli kolejności wykonywania. Wiadomo np., że mnożenie jest „siłniejsze” od dodawania, zatem najpierw mnożymy, potem dodajemy (kolejność tę można zmienić, stosując nawiasy okrągłe, dokładnie w taki sam sposób, w jaki zmienia się kolejność działań w matematyce). W JavaScriptie jest podobnie — siła każdego operatora jest ścisłe określona. Przedstawiono to w tabeli 2.17. Im wyższa pozycja w tabeli, tym wyższy priorytet operatora. Operatory znajdujące się na jednym poziomie (w jednym wierszu) mają ten sam priorytet⁵.

Tabela 2.17. Priorytety operatorów

L.p.	Operatory	Symbol
1	indeks tablicy, wywołanie funkcji	[], ()
2	inkrementacja i dekrementacja, ustalenie znaku, negacja bitowa i logiczna, utworzenie obiektu, ustalenie typu zmiennej, usunięcie składowej	++, --, +, -, ~, !, new, typeof, delete
3	mnożenie, dzielenie, reszta z dzielenia	*, /, %
4	dodawanie, odejmowanie	+, -
5	przesunięcie bitowe w lewo, w prawo, w prawo z wypełnieniem zerami	<<, >>, >>>
6	mniejsze, większe, mniejsze lub równe, większe lub równe, porównanie typów	<, >, <=, >=
7	równie, różne	==, !=
8	iloczyn bitowy	&
9	bitowa różnica symetryczna	^
10	suma bitowa	
11	iloczyn logiczny	&&
12	suma logiczna	
13	warunkowy	? :
14	operatory przypisania	=, +=, -=, *=, /=, %=, ^=, =, <<=, >=, >>=
15	rozdzielanie wyrażeń	,

⁵ Tabela uwzględnia również operatory, które nie były omawiane w książce.

Operacje na ciągach znaków

W części opisującej operatory arytmetyczne poznaliśmy operator dodawania +. W wielu skryptach używaliśmy go jednak w zupełnie innym celu — do łączenia ciągów znaków. Wiemy, że operacja:

"abc" + "def"

daje w wyniku ciąg:

"abcdef"

Można też dodać ciąg znaków do zmiennej; korzystaliśmy z konstrukcji typu:

```
var str = "abc";
str = str + "def";
```

Jest to postępowanie poprawne, gdyż w rzeczywistości + oznacza dwa różne typy operacji. Pierwsza z nich to arytmetyczne dodawanie, a druga — łączenie (czyli inaczej, konkatenacja)łańcuchów znakowych (ciągów znaków). Musimy jednak dobrze zdawać sobie sprawę, kiedy mamy do czynienia z jednym, a kiedy z drugim rodzajem operacji. Sytuacje typu:

5 + 8

czy:

"To " + "jest"

są jasne. W pierwszym przypadku mamy dodawanie arytmetyczne, w drugim — łączenie ciągów znaków. Operacja:

"123" + "456"

też nie powinna przysporzyć kłopotu. Zauważmy, że oba operandy (argumenty operatora +) są ujęte w znaki cudzysłowu. To oznacza, że są to ciągi znaków i mamy do czynienia z konkatenacją. Zatem nie jest to dodawanie arytmetyczne, ale łączeniełańcuchów. Wynikiem będzie więc ciąg:

"123456"

Musimy też rozpatrzyć sytuację, gdy jednym argumentem jest ciąg znaków, a drugim wartość liczbową, np.:

"abc" + 5

Oczywiście, taka operacja arytmetyczna nie jest możliwa (nie można dodać liczby do ciągu znaków), zatem zostanie potraktowana jak łączeniełańcuchów znakowych. To oznacza, że argument arytmetyczny (wartość 5) zostanie potraktowany jako ciąg znaków (przekonwertowany na ciąg znaków) i operacja ta zostanie potraktowana jak:

"abc" + "5"

a jej wynikiem będzie:

"abc5"

Tu nie ma też znaczenia kolejność występowania operandów. Działanie:

5 + "abc"

zostanie potraktowane jak:

"5" + "abc"

Do tego tematu powrócimy jeszcze w dalszej części książki.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 3.1.

W kodzie z listingu 2.13 usuń fragmenty dodające na końcu każdego wiersza znaczek
. W jaki sposób zostaną wtedy wyświetlane dane i dlaczego?

Ćwiczenie 3.2.

Zmień kod z listingu 2.14 tak, aby do dodawania danych do zmiennej str zamiast operatora = był używany +=.

Ćwiczenie 3.3.

Napisz skrypt prezentujący na stronie wyniki działania sumy bitowej, iloczynu bitowego i bitowej różnicy symetrycznej.

Lekcja 4. Instrukcje warunkowe

Podczas pisania skryptów często spotkamy się z sytuacją, w której konieczne będzie zbadanie pewnego warunku (np. czy wartość zmiennej jest większa od 0) i — zależnie od tego, czy jest prawdziwy, czy nie — wykonanie różnych operacji. Najprostszym przykładem może być wyświetlanie różnych tekstów, w zależności od tego, czy wybrana zmienna ma wartość dodatnią. Na takie rozwidlenie kodu programu pozwalają instrukcje warunkowe, które omówimy w tej właśnie lekcji. W JavaScriptie, tak jak w innych popularnych językach skryptowych oraz klasycznych językach programowania, do badania warunków służą różne postacie instrukcji if.

Instrukcja if

W najprostszym przypadku instrukcja if ma postać:

```
if (warunek){  
    //Instrukcje wykonywane,  
    //gdy warunek jest prawdziwy.  
}
```

Zapis ten należy rozumieć tak: „Jeśli warunek (umieszczony w nawiasie okrągłym za słowem `if`) jest prawdziwy, wykonaj blok instrukcji znajdujący się pomiędzy znakami nawiasu klamrowego”. Za instrukcją `if`, czyli za znakiem `}` kończącym blok `if`, średnik może występować, ale nie musi (z reguły jest pomijany). Zwrócmy też uwagę na zastosowany tu sposób formatowania kodu. Znak `{` otwierający blok instrukcji przy-należnych do `if` jest umieszczony za warunkiem. Można go jednak z powodzeniem umieścić w nowym wierszu, czyli:

```
if (warunek)
{
    //Instrukcje wykonywane,
    //gdy warunek jest prawdziwy.
}
```

Obie formy są równoważne i należy stosować tę, która wydaje się bardziej czytelna. To zależy od indywidualnych preferencji programisty. W książce będzie stosowana pierwsza z zaprezentowanych form.

Gdy w bloku instrukcji `if` miałaby się znaleźć tylko jedna instrukcja, czyli miałby on postać:

```
if (warunek){
    instrukcja;
}
```

dopuszczalne jest pominięcie nawiasu klamrowego, jednak obligatoryjny jest wtedy średnik kończący. Zatem prawidłowy jest zapis:

```
if (warunek)
    instrukcja;
```

bądź też:

```
if (warunek) instrukcja;
```

Do skonstruowania warunku wykorzystywane są operatory relacyjne przedstawione w lekcji 3. Zobaczmy, jak taka konstrukcja będzie wyglądała w praktyce. Przykładowy kod wykorzystujący instrukcję warunkową `if` został zaprezentowany na listingu 2.16.

Listing 2.16. Przykład użycia instrukcji if

```
var liczba = -1;

if(liczba < 0){
    str = "Zmienna liczba jest mniejsza od 0.";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Powstała tu zmienna `liczba` i została jej przypisana wartość `-1`. Za pomocą instrukcji warunkowej `if` został zbadany warunek `liczba < 0`. Ten warunek jest prawdziwy, skoro bowiem zawartość zmiennej `liczba` to `-1`, jest ona niewątpliwie mniejsza od `0`.

Dlatego też zostanie wykonany blok instrukcji umieszczony pomiędzy znakami nawiąsu klamrowego. W tym przypadku jest to tylko jedna instrukcja, przypisująca zmiennej str ciąg znaków. A zatem po uruchomieniu skryptu na ekranie pojawi się napis Zmienna liczba jest mniejsza od 0.

Co się jednak stanie, jeśli zmieniąca wartość zmienność zmiennej str będzie miała nieujemną wartość? Wtedy na ekranie nic nie zobaczymy. Co więcej, choć nie będzie tego widać bezpośrednio, w skrypcie wystąpi błąd. Zauważmy bowiem, że jeśli nie zostanie wykonany blok instrukcji warunkowej if, to nie zostanie zdefiniowana zmienność zmiennej str. Tymczasem jest ona używana w ostatnim wierszu skryptu. Problem można rozwiązać, definiując tę zmienność przed instrukcją warunkową, np.:

```
var liczba = -1;
var str = "";
if(liczba < 0){
    str = "Zmienna liczba jest mniejsza od 0.";
    //...itd.
```

Oczywiście, wtedy przy ujemnej wartości zmiennej nadal na ekranie nic się nie pojawi, ale skrypt będzie pracował bezbłędnie. Przy okazji poznaliśmy ciekawą konstrukcję. Cóż bowiem zostało przypisane zmiennej str w poniższym wierszu?

```
var str = "";
```

Znaki cudzysłowu sugerują, że jest to ciąg znaków, jednak między nimi żadne znaki nie występują. Taką konstrukcję nazywamy pustym ciągiem znaków (choć może brzmi dziwnie, przydaje się przy pisaniu skryptów). Oznacza to, że typem zmiennej str jest typ łańcuchowy, jednak jest ona pusta, czyli nie zostały w niej zapisane żadne znaki (choć jako programiści powinniśmy powiedzieć, że został w niej zapisany pusty ciąg znaków).

Jeżeli chcemy, aby informacja o stanie zmiennej pojawiała się w obu przypadkach (czyli gdy jest ujemna bądź nieujemna), możemy zastosować dwie instrukcje warunkowe if, jak zostało to zaprezentowane na listingu 2.17.

Listing 2.17. Instrukcje if rozpatrujące dwa warunki

```
var liczba = 1;

if(liczba < 0){
    str = "Zmienna liczba jest mniejsza od 0. ";
}
if(liczba >= 0){
    str = "Zmienna liczba nie jest mniejsza od 0. ";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Pierwsza instrukcja `if` bada, czy zmienna `liczba` ma wartość mniejszą od 0. Ma ona taką samą postać jak w poprzednim przykładzie. Druga instrukcja `if` bada, czy zmienna `liczba` jest większa od 0 lub mu równa. W obu przypadkach, jeżeli badany warunek jest prawdziwy, zmiennej `str` jest przypisywany odpowiedni komunikat, który pojawi się na ekranie.

Instrukcja `if...else`

W przykładzie zaprezentowanym na listingu 2.17 zostały użyte dwie instrukcje `if` do sprawdzenia dwóch przeciwnych, wykluczających się wzajemnie warunków. Wykluczających się, ponieważ zmienna `liczba` mogła być albo mniejsza, albo większa lub równa 0 (czyli nie mniejsza od 0). W takiej sytuacji można jednak użyć również rozbudowanej instrukcji `if`, której ogólna postać jest następująca:

```
if (warunek){  
    //instrukcje wykonywane, gdy warunek jest prawdziwy  
}  
else{  
    //instrukcje wykonywane, gdy warunek jest fałszywy  
}
```

Jak widać, został dodany blok `else`, który jest wykonywany, gdy `warunek` jest fałszywy. Całą konstrukcję należy zatem rozumieć tak: „Jeżeli warunek jest prawdziwy, wykonaj instrukcje występujące po `if` (w bloku `if`); w przeciwnym przypadku wykonaj instrukcje występujące po `else` (w bloku `else`)”.

A więc w przykładzie z listingu 2.17 zamiast dwóch instrukcji `if` można by zastosować jedną instrukcję `if...else`. Byłoby to rozwiązanie lepsze i czytelniejsze. Kod powinien zatem wyglądać tak, jak na listingu 2.18.

Listing 2.18. Użycie instrukcji `if...else`

```
var liczba = 1;  
  
if(liczba < 0){  
    str = "Zmienna liczba jest mniejsza od 0.>";  
}  
else{  
    str = "Zmienna liczba nie jest mniejsza od 0.>";  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Ponieważ w powyższym przykładzie zarówno w bloku `if`, jak i w bloku `else` występuje tylko jedna instrukcja (przypisanie zmiennej `str` ciągu znaków), zgodnie z tym, co zostało napisane wcześniej, w obu przypadkach możliwe jest pominięcie nawiasu klamrowego. Instrukcja `if...else` mogłaby zatem przyjąć postać:

```
if(liczba < 0)  
    str = "Zmienna liczba jest mniejsza od 0.>";  
else  
    str = "Zmienna liczba nie jest mniejsza od 0.>";
```

Instrukcja if...else if

Kolejna wersja instrukcji if pozwala na badanie wielu warunków. Otóż, po if może wystąpić wiele dodatkowych bloków else if. Schematyczna postać takiej konstrukcji to:

```
if (warunek1){  
    instrukcje1;  
}  
else if (warunek2){  
    instrukcje2;  
}  
else if (warunek3){  
    ..instrukcje3;  
}  
...  
else if (warunekN){  
    ..instrukcjeN;  
}  
else{  
    ..instrukcjeM;  
}
```

Oznacza to: „Jeżeli warunek1 jest prawdziwy, to zostaną wykonane instrukcje1. W przeciwnym wypadku, jeżeli jest prawdziwy warunek2, zostaną wykonane instrukcje2. Jeżeli natomiast jest prawdziwy warunek3, zostaną wykonane instrukcje3 itd. Jeżeli żaden z warunków nie będzie prawdziwy, zostaną wykonane instrukcjeM”. Ostatni blok else jest jednak opcjonalny i nie musi być stosowany.

Taka konstrukcja może być wykorzystana np. wtedy, kiedy chcemy wykonać wiele różnych instrukcji zależnych od stanu zmiennej. Sposób, w jaki należy to zrobić, zobrazowano w skrypcie widocznym na listingu 2.19.

Listing 2.19. Użycie instrukcji if...else if

```
var liczba = 30;  
  
if(liczba == 0){  
    str = "Wartość zmiennej liczba jest równa 0.:";  
}  
else if(liczba == 10){  
    str = "Wartość zmiennej liczba jest równa 10.:";  
}  
else if(liczba == 20){  
    str = "Wartość zmiennej liczba jest równa 20.:";  
}  
else{  
    str = "Wartość zmiennej liczba nie jest równa ani 0, ani 10, ani 20.";  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Przedstawiona w skrypcie złożona instrukcja warunkowa bada po kolej warunki: `liczba == 0`, `liczba == 10`, `liczba == 20`, czyli sprawdza, czy zmienna `liczba` ma wartość 0, 10 lub 20. Jeśli którykolwiek z tych warunków jest prawdziwy, zmiennej str przypisywany jest komunikat informujący o wartości zmiennej. W przypadku, kiedy wszystkie warunki są fałszywe, jest wykonywany blok `else`, czyli instrukcja przypisująca zmiennej str komunikat, że zmienna `liczba` nie jest równa ani 0, ani 10, ani 20.

Zagnieżdżanie instrukcji warunkowych

Zarówno w bloku `if`, jak i w `else` mogą wystąpić dowolne inne instrukcje. Oznacza to zatem, że można tam umieścić kolejne instrukcje `if`, a więc mogą być one zagnieżdżane. Schematycznie taka konstrukcja będzie wyglądała następująco:

```
if (warunek1){
    if (warunek2){
        instrukcje1:
    }
    else{
        instrukcje2:
    }
}
else{
    if (warunek3){
        instrukcje3:
    }
    else{
        instrukcje4:
    }
}
```

Taka struktura ma następujące znaczenie: „`instrukcje1` zostaną wykonane, kiedy prawdziwe są `warunek1` i `warunek2`, `instrukcje2` — kiedy prawdziwy jest `warunek1`, a fałszywy — `warunek2`, `instrukcje3` — kiedy fałszywy jest `warunek1` i prawdziwy jest `warunek3`, a `instrukcje4`, kiedy są fałszywe `warunek1` i `warunek3`”. Oczywiście, nie musimy się ograniczać do przedstawionych tu dwóch poziomów zagnieżdżenia — może ich być dużo więcej.

Użyjmy zagnieżdżonej instrukcji `if` w skrypcie wykonującym bardzo konkretne zadanie. Będzie obliczał rozwiązania równania kwadratowego o zadanych parametrach. Przypomnijmy, że równanie takie ma ogólną postać: $A*x^2+B*x+C = 0$, gdzie A , B i C to parametry równania. Równanie ma rozwiązanie w zbiorze liczb rzeczywistych, jeśli parametr Δ (delta) równy $B^2 - 4*A*C$ jest większy od 0 lub mu równy. Jeśli Δ równa się 0, mamy jedno rozwiązanie równe $-B/(2*A)$, jeśli Δ jest większa od 0, mamy dwa rozwiązania: $x_1 = (-B + \sqrt{\Delta})/(2*A)$ i $x_2 = (-B - \sqrt{\Delta})/(2*A)$. Taka liczba warunków doskonale nadaje się do przećwiczenia działania instrukcji `if...else`. Jedyną niedogodnością skryptu będzie to, że parametry A , B i C będą musiały być wprowadzone bezpośrednio w jego kodzie, nie znamy bowiem jeszcze sposobu na przekazanie danych z przeglądarki (temat ten zostanie omówiony w rozdziale 4.). Skrypt wykonujący przedstawione zadania został zaprezentowany na listingu 2.20.

Listing 2.20. Zagnieżdzona instrukcja if i równania kwadratowe

```
//deklaracje zmiennych
var A = 1;
var B = 1;
var C = -2;
var str = "";

//wyświetlenie parametrów równania
str += "Parametry równania: <br />";
str += "A = " + A + ", B = " + B + ", C = ";
str += C + " <br />";

//sprawdzenie, czy jest to równanie kwadratowe
if (A == 0){
    //A jest równe 0, równanie nie jest kwadratowe
    str += "To nie jest równanie kwadratowe: A = 0!";
}
else{
    //A jest różne od 0, równanie jest kwadratowe

    //obliczenie delty
    delta = B * B - 4 * A * C;

    //jeśli delta mniejsza od 0
    if (delta < 0){
        str += "Delta < 0<br>";
        str += "To równanie nie ma rozwiązań w zbiorze ";
        str += "liczb rzeczywistych!";
    }
    //jeśli delta większa lub równa 0
    else{
        //jeśli delta jest równa 0
        if (delta == 0){
            //obliczenie wyniku
            wynik = - B / 2 * A;
            str += "Rozwiązań: x = " + wynik;

        }
        //jeśli delta jest większa od 0
        else{
            //obliczenie wyników
            wynik = (- B + Math.sqrt(delta)) / 2 * A;
            str += "Rozwiązań: x1 = " + wynik;
            wynik = (- B - Math.sqrt(delta)) / 2 * A;
            str += ", x2 = " + wynik;
        }
    }
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Dla ułatwienia orientacji w kodzie skryptu poszczególne jego fragmenty zostały oznaczone komentarzami. Zaczynamy od zadeklarowania i zainicjalizowania trzech zmiennych: A, B i C, które odzwierciedlają parametry równania, oraz wyświetlenia ich wartości na ekranie (przez przypisanie ich wartości zmiennej str). Za pomocą instrukcji if sprawdzamy, czy zmienna A jest równa 0. Jeśli tak, oznacza to, że równanie nie jest kwadratowe, jest więc wyświetlana stosowna informacja i jest to koniec działania skryptu. Jeśli jednak A jest różne od 0, możemy przystąpić do obliczenia delty. Wynik obliczeń przypisujemy zmiennej delta. Kolejny krok to sprawdzenie, czy delta nie jest przypadkiem mniejsza od 0. Jeśli jest, oznacza to, że równanie nie ma rozwiązań w zbiorze liczb rzeczywistych, wyświetlamy więc odpowiedni komunikat (przypomnijmy raz jeszcze, że jako wyświetlenie komunikatu traktujemy przypisanie go zmiennej str) — po jego wyświetleniu skrypt kończy działanie.

Kiedy jednak delta nie jest mniejsza od 0, przystępujemy do sprawdzenia kolejnych warunków. Jeśli jest równa 0, można od razu obliczyć rozwiązanie równania ze wzoru $-B / (2 * A)$. Wynik tych obliczeń przypisujemy zmiennej pomocniczej wynik i wyświetlamy komunikat z rozwiązaniem na ekranie.

Gdy delta jest większa od 0, istnieją dwa pierwiastki (rozwiązań) równania. Obliczamy je w linach:

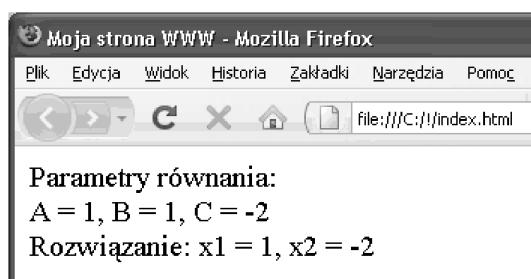
```
wynik = (- B + Math.sqrt(delta)) / 2 * A;
```

oraz:

```
wynik = (- B - Math.sqrt(delta)) / 2 * A;
```

Ostateczny rezultat obliczeń pojawi się w oknie przeglądarki, co widać na rysunku 2.6. Nieznana instrukcja Math.sqrt(delta) powoduje obliczenie pierwiastka kwadratowego (drugiego stopnia) z wartością zawartą w zmiennej delta. O instrukcjach wykonujących operacje matematyczne będzie jeszcze mowa w dalszej części książki. W ramach ćwiczeń do samodzielnego wykonania warto natomiast przetworzyć zaprezentowany skrypt tak, aby korzystał z instrukcji if...else if.

Rysunek 2.6.
Wynik działania skryptu obliczającego równania kwadratowe



Złożone wyrażenia warunkowe

Warunki, a dokładniej wyrażenia warunkowe, stosowane w dowolnej postaci instrukcji if mogą być bardziej złożone niż dotychczas prezentowane. Mogą się składać z wielu członów połączonych operatorami logicznymi. Założymy, że mamy wykonać

pewne czynności, np. wyświetlić napis, kiedy dana zmienna jest większa od 0, ale jednocześnie mniejsza od 10. Istnieją dwa sposoby realizacji tego zadania. Pierwszy to użycie zagnieżdzonej instrukcji if w postaci:

```
if(liczba > 0){  
    if(liczba < 10){  
        str = "Liczba jest większa od 0 i mniejsza od 10.";  
    }  
}
```

Drugi to wykorzystanie pojedynczej instrukcji if ze złożonym wyrażeniem warunkowym. Jak zbudować takie wyrażenie? Na takiej samej zasadzie jak na szkolnych lekcjach matematyki. Skoro interesuje nas sytuacja, kiedy liczba jest większa od 0 i mniejsza od 10, trzeba skorzystać z iloczynu logicznego, który w JavaScriptie jest wyrażany za pomocą operatora &&. A zatem wyrażenie warunkowe będzie miało postać:

```
(liczba > 0) && (liczba < 10)
```

Cała instrukcja if będzie więc wyglądała następująco:

```
if((liczba > 0) && (liczba < 10)){  
    str = "Liczba jest większa od 0 i mniejsza od 10.";  
}
```

Jeśli zajrzymy na chwilę do tablicy 2.17 przedstawiającej priorytety operatorów, przekonamy się, że operatory relacyjne $>$ i $<$ mają większy priorytet (są silniejsze) niż operator iloczynu logicznego $\&\&$. Wynika z tego, że można pominąć nawiasy w wyrażeniu warunkowym, które równie dobrze może mieć postać:

```
liczba > 0 && liczba < 10
```

Jest to w pełni dopuszczalne, aczkolwiek nie zaleca się tego rozwiązania, gdyż zaciemnia kod i utrudnia jego analizę, szczególnie w przypadku bardziej złożonych wyrażeń.

Wykonajmy jeszcze jeden przykład. Tym razem napis powinien zostać wyświetlony, kiedy wartość zmiennej liczba jest większa od 0, ale różna od 5 i różna od 10, lub też gdy jest równa -8. Możemy tu wyróżnić 4 wyrażenia składowe:

- ◆ $liczba > 0$,
- ◆ $liczba \neq 5$,
- ◆ $liczba \neq 10$,
- ◆ $liczba == -8$,

Należy je połączyć za pomocą operatorów logicznych, tak jak poniżej:

```
(liczba > 0) && (liczba != 5) && (liczba != 10)) || (liczba == -8)
```

Instrukcja if przyjmie zatem postać:

```
if(((liczba > 0) && (liczba != 5) && (liczba != 10)) || (liczba == -8)){  
    str += "Zmienna liczba spełnia wszystkie warunki."  
}
```

Instrukcja wyboru switch

Instrukcja wyboru switch (nazywana również instrukcją switch...case) pozwala w wygodny sposób sprawdzić ciąg warunków i wykonać różne instrukcje, w zależności od wyników porównywania. Ma ogólną postać:

```
switch(wyrażenie){
    case wartość1 :
        instrukcje1;
        break;
    case wartość2 :
        instrukcje2;
        break;
    case wartość3 :
        instrukcje3;
        break;
    default :
        instrukcje4;
}
```

Należy rozumieć ją następująco: „Sprawdź wartość wyrażenia *wyrażenie*, jeśli wynikiem jest *wartość1*, wykonaj *instrukcje1* i przerwij wykonywanie bloku switch (instrukcja break). Jeśli wynikiem jest *wartość2*, wykonaj *instrukcje2* i przerwij wykonywanie bloku switch, jeśli wynikiem jest *wartość3*, wykonaj *instrukcje3* i przerwij wykonywanie bloku switch. Jeśli nie zachodzi żaden z wymienionych przypadków, wykonaj *instrukcje4* i zakończ blok switch”. Blok default jest opcjonalny i może zostać pominięty.

Łatwo zauważyc, że jest to odpowiednik złożonej instrukcji if...else if w postaci:

```
if(wyrażenie == wartość1){
    instrukcje1;
}
else if(wyrażenie == wartość2){
    instrukcje2;
}
else if(wyrażenie == wartość3){
    instrukcje3;
}
else{
    instrukcje4;
}
```

Zobaczmy jednak, jak wykorzystać instrukcję wyboru w działającym skrypcie. Przykład znajduje się na listingu 2.21.

Listing 2.21. Działanie instrukcji wyboru switch

```
var liczba = 10;

switch(liczba){
    case 10 :
        str = "Zmienna liczba = 10..";
        break;
```

```
case 20 :  
    str = "Zmienna liczba = 20.;"  
    break;  
default :  
    str = "Zmienna liczba nie jest równa ani 10, ani 20."  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Na początku została utworzona zmienna `liczba` o wartości 10. Instrukcja `switch` najpierw oblicza wartość wyrażenia występującego w nawiasie okrągłym. Ponieważ w tym przypadku jako wyrażenie występuje nazwa zmiennej, wartością wyrażenia staje się wartość tej zmiennej. Wartość jest porównywana do wartości występujących po słowach `case`, czyli 10 i 20. Jeśli zgodność zostanie stwierdzona, wykonane będą instrukcje występujące w danym bloku `case`. Jeśli nie uda się dopasować wartości wyrażenia do żadnej z wartości występujących po słowach `case`, jest wykonywany blok `default`. Ponieważ wartością wyrażenia (zmiennej) jest 10, zgodność jest stwierdzana już w pierwszym przypadku i zostanie wykonana instrukcja przypisująca zmiennej `str` napis `Zmienna liczba = 10`. Występująca po tym przypisaniu instrukcja `break` przerwuje wykonywanie bloku `case`.

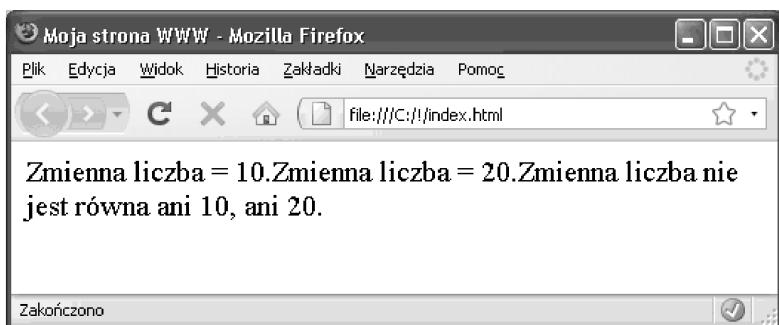
Na instrukcję `break` należy zwrócić szczególną uwagę. Jej przypadkowe pominięcie może doprowadzić do nieoczekiwanych wyników i błędów w skrypcie. Aby przekonać się, w jaki sposób działa instrukcja `switch` bez instrukcji `break`, zmodyfikujemy skrypt z listingu 2.21, usuwając z niego wszystkie instrukcje `break`. Powstanie wtedy kod widoczny na listingu 2.22.

Listing 2.22. Pominięcie instrukcji `break`

```
var liczba = 10;  
var str = "";  
  
switch(liczba){  
    case 10 :  
        str += "Zmienna liczba = 10.;"  
    case 20 :  
        str += "Zmienna liczba = 20.;"  
    default :  
        str += "Zmienna liczba nie jest równa ani 10, ani 20."  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Jeśli go uruchomimy, w przeglądarce zobaczymy widok zaprezentowany na rysunku 2.7. Skrypt wyraźnie nie spełnia swojego zadania. Zmienna nie może przecież jednocześnie spełniać trzech przeciwnych warunków. Jak więc działa przedstawiony kod? Otóż, jeśli w którymś z bloków (przypadków) `case` zostanie wykryta zgodność z wyrażeniem występującym za `switch`, zostaną wykonane wszystkie dalsze instrukcje

Rysunek 2.7.
Wynik pominięcia
instrukcji break



aż do napotkania instrukcji `break`⁶ lub dotarcia do końca instrukcji `switch`. W kodzie z listingu 2.22 zgodność jest stwierdzana już w pierwszym bloku `case`, jest więc wykonywana znajdująca się w nim instrukcja przypisania. Ponieważ jednak w bloku tym nie ma instrukcji `break`, są wykonywane instrukcje znajdujące się w kolejnym bloku `case (case 20)`. W tym bloku również brakuje `break`, a zatem są wykonywane instrukcje znajdujące się po słowie `default`. Tym samym wykonane zostaną wszystkie znajdujące się w kodzie instrukcje dopisujące dane do zmiennej `str`. Dlatego też na ekranie pojawiają się wszystkie zdefiniowane w kodzie komunikaty.

Operator warunkowy

Operator warunkowy pozwala na ustalenie wartości wyrażenia, w zależności od prawdziwości danego warunku. Ma postać:

`warunek ? wartość1 : wartość2`

Oznacza to: „Jeśli warunek jest prawdziwy, podstaw za wartość całego wyrażenia `wartość1`, w przeciwnym przypadku za wartość wyrażenia podstaw `wartość2`”. Spójrzmy na kod z listingu 2.23 i zastanówmy się, jaka wartość zostanie przypisana zmiennej wynik, a tym samym wyświetlona na stronie?

Listing 2.23. Użycie operatora warunkowego

```
var liczba = 100;
var wynik = (liczba < 0) ? -1 : 1;

str = wynik;
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Oczywiście, najważniejsza jest tu linia `var wynik = (liczba < 0) ? -1 : 1;`. Po lewej stronie operatora przypisania — znajduje się zmienna (`wynik`), natomiast po stronie prawej — wyrażenie warunkowe, czyli linia ta oznacza: „Przypisz zmiennej `wynik` wartość wyrażenia warunkowego”. Jak jest ta wartość? Trzeba przeanalizować samo wyrażenie: `(liczba < 0) ? -1 : 1`. Oznacza ono, zgodnie z tym, co zostało napisane

⁶ Dokładniej — dowolnej instrukcji powodującej opuszczenie bloku `switch`.

w poprzednim akapicie: „Jeżeli wartość zmiennej `liczba` jest mniejsza od 0, przypisz wyrażeniu wartość -1, w przeciwnym przypadku (zmienna `liczba` większa lub równa 0) przypisz wyrażeniu wartość 1”. Ponieważ zmiennej `liczba` przypisaliśmy wcześniej wartość 100, wartością całego wyrażenia będzie 1 i ta właśnie wartość zostanie przypisana zmiennej wynik.

W tym miejscu ponownie zatrzymy się do tabeli 2.17. Z informacji tam podanych wynika, że operator warunkowy ma priorytet mniejszy niż operatory relacyjne. W związku z tym, nawiasy okrągłe, wprowadzone do wyrażenia w celu zwiększenia czytelności zapisu, mogą zostać pominięte i może mieć ono również postać:

```
liczba < 0 ? -1 : 1;
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 4.1.

Zastanów się, jaki będzie efekt działania (i dlaczego) skryptu 2.16, jeżeli zmienna `liczba` będzie nieujemna, a przed instrukcją warunkową `if` zostanie zadeklarowana zmienna `str`, ale nie zostanie jej przypisana żadna wartość.

Ćwiczenie 4.2.

Zmień treść skryptu 2.16 w taki sposób, aby pozostała w nim tylko jedna instrukcja warunkowa `if` (nie `if...else`), a w przypadku gdy zmienna `liczba` zawiera wartość nieujemną, wyświetlana była prawidłowa informacja.

Ćwiczenie 4.3.

Na podstawie przykładu z listingu 2.20 napisz skrypt obliczający rozwiązania równania kwadratowego, który będzie korzystał z instrukcji `if...else if`.

Ćwiczenie 4.4.

Na podstawie kodu z listingu 2.21 napisz skrypt zawierający instrukcję wyboru `switch`, rozpoznający dwie sytuacje — kiedy zmienna `liczba` jest równa 10 lub 20 oraz kiedy jest różna od 10 i od 20.

Ćwiczenie 4.5.

Napisz złożone wyrażenie warunkowe określające stan zmiennej `liczba`. Użyj instrukcji warunkowej do wyświetlenia komunikatu, gdy wartość tej zmiennej jest większa od -15 i mniejsza od 24, ale różna od 0 i 5, bądź równa 52 lub 83.

Lekcja 5. Pętle

Pętle to takie konstrukcje języka, które pozwalają na wielokrotne wykonywanie tych samych instrukcji. Jeśli np. chcemy 20 (czy też 100 albo 1000) razy wyświetlić na ekranie napis, możemy w tym celu użyć 20 przypisań zmiennej str lub (tylko w HTML-u) 20 instrukcji document.write, ale byłoby to niewątpliwie niezbyt wygodne rozwiązanie. Co więcej, bardzo często spotykamy się z sytuacją, gdy liczba powtórzeń danej instrukcji nie jest z góry znana i wynika z obliczeń wykonywanych przez skrypt bądź też zależy od decyzji użytkownika strony WWW. Przykładowo liczba wyświetlanych na stronie napisów może być zależna od tego, jaką wartość użytkownik strony wprowadzi do pewnego pola formularza⁷. Co robić w takich sytuacjach? Użyć pętli. W JavaScriptie mamy do dyspozycji następujące ich rodzaje:

- ◆ for,
- ◆ for...in,
- ◆ while,
- ◆ do...while.

Pętla for

Pętla typu for ma ogólną postać:

```
for(wyrażenie początkowe; wyrażenie warunkowe; wyrażenie modyfikujące){
    instrukcje do wykonania
}
```

I tak *wyrażenie początkowe* jest stosowane do zainicjalizowania zmiennej używanej jako licznik liczby wykonania pętli; *wyrażenie warunkowe* określa warunek, jaki musi być spełniony, aby dokonać kolejnego przejścia w pętli, natomiast *wyrażenie modyfikujące* jest zwykle wykorzystywane do modyfikacji zmiennej będącej licznikiem. Sposób działania takiej pętli najłatwiej pokazać na konkretnym przykładzie. Jest on widoczny na listingu 2.24.

Listing 2.24. Użycie pętli for

```
var str = "";

for(i = 0; i < 10; i++){
    str += "Ten napis się powtarza...";
    str += "<br />";
}

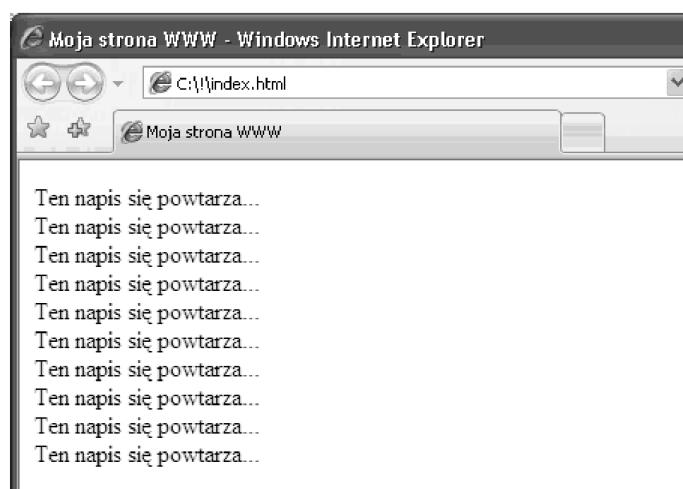
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

⁷ Sposobami obsługi formularzy i innych elementów strony zajmiemy się w rozdziale 3.

Na początku zadeklarowana została zmienna str i został jej przypisany pusty ciąg znaków. Do tej zmiennej będą dodawane kolejne ciągi znaków, tak aby wszystkie mogły pojawić się na stronie WWW. Za deklaracją zmiennej została umieszczona pętla for. Znaczenie tej konstrukcji jest następujące: „Utwórz zmienną i i przypisz jej wartość 0 ($i = 0$); następnie — tak długo, jak długo wartość i jest mniejsza od 10 ($i < 10$) — wykonuj instrukcję znajdującej się wewnątrz pętli (pomiędzy znakami nawiasu klamrowego) oraz zwiększaj i o jeden ($i++$)”. Ponieważ każde wykonanie pętli to dodanie do zmiennej str ciągu znaków Ten napis się powtarza... oraz znacznika `
`, na ekranie pojawi się 10 takich napisów, każdy w osobnym wierszu, co widać na rysunku 2.8. Tu uwaga dotycząca terminologii: każde kolejne wykonanie instrukcji wewnętrznej pętli nazywamy kolejnym *przebiegiem pętli* lub kolejną *iteracją*. Z kolei zmienną sterującą pętlą, czyli tą, od której zależy jej wykonanie (w tym przypadku chodzi o zmienną i), nazywamy *zmienną iteracyjną* pętli.

Rysunek 2.8.

Efekt użycia pętli for do wielokrotnego wyświetlenia napisu



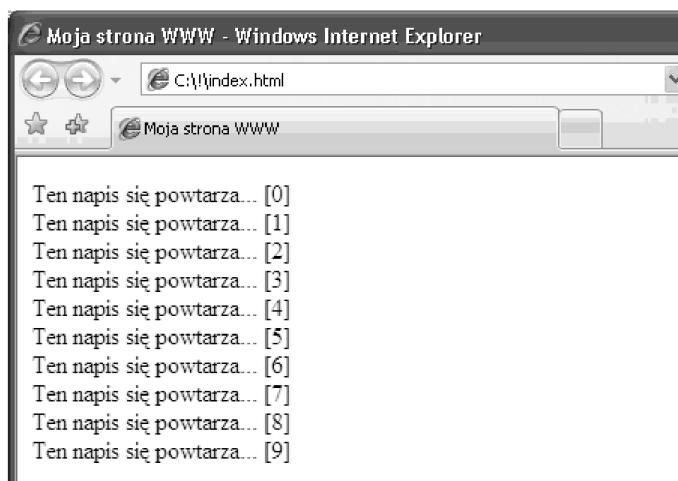
Jeśli chcemy zobaczyć, jak zmienia się stan zmiennej i w trakcie kolejnych przebiegów (iteracji), możemy zmodyfikować instrukcję wyświetlającą napis, tak aby podawała również i tę informację. Wystarczy niewielka zmiana:

```
for(i = 0; i < 10; i++){
    str += "Ten napis się powtarza... [" + i + "]";
    str += "<br />";
}
```

Po jej wprowadzeniu w każdej linii będzie również wyświetlana wartość i (czyli numer kolejnego przebiegu pętli), jak zostało to przedstawione na rysunku 2.9. Zwrócmy uwagę, że wartość ta zmienia się od 0 do 9, ponieważ początkowym stanem zmiennej i było 0.

Rysunek 2.9.

Prosta modyfikacja pętli pozwala podejrzeć stan zmiennej iteracyjnej



Pętla while

Pętla typu while służy, podobnie jak for, do wykonywania powtarzających się czynności. Pętlę for najczęściej wykorzystuje się, kiedy liczba powtarzanych operacji jest znana, natomiast pętlę while, kiedy liczby powtórzeń nie znamy, a zakończenie pętli jest uzależnione od spełnienia pewnego warunku. Taki podział jest jednak dosyć umowny, gdyż oba typy pętli można zapisać w taki sposób, aby były swoimi funkcjonalnymi odpowiednikami. Ogólna postać pętli while wygląda następująco:

```
while (warunek){
    instrukcje;
}
```

Oto jej znaczenie: „Dopóki *warunek* jest prawdziwy, wykonuj *instrukcje*”. Użyjmy więc takiej pętli do wyświetlenia serii powtarzających się napisów. To zadanie realizuje skrypt widoczny na listingu 2.25.

Listing 2.25. Użycie pętli while

```
var str = "";
i = 0;
while(i < 10){
    str += "Ten napis się powtarza... ";
    str += "<br />";
    i++;
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

W tym przypadku instrukcje znajdujące się we wnętrzu pętli wykonywane są dopóty, dopóki wartość zmiennej *i* jest mniejsza od 10, warunkiem zakończenia pętli jest bo-wiem *i* < 10. Ponieważ początkową wartością *i* jest 0, a wewnątrz pętli znajduje się instrukcja zwiększająca w każdym jej przebiegu wartość *i* o 1 (*i++*), mamy pewność, że pętla się zakończy. Pisząc pętlę *while*, trzeba zwracać szczególną uwagę na warunek zakończenia, gdyż łatwo napisać pętlę wykonującą się w nieskończoność.

Oczywiście, również w tym przypadku (podobnie jak przy pętli *for*) możemy pokusić się o wyświetlenie stanów zmiennej iteracyjnej *i*. Wystarczy modyfikacja w postaci:

```
while(i < 10){
    str += "Ten napis się powtarza... [" + i + "]";
    str += "<br />";
    i++;
}
```

Efekt będzie taki sam jak na rysunku 2.9.

Zmodyfikujmy teraz nasz przykład w taki sposób, aby modyfikacja zmiennej *i* odbywała się w wyrażeniu warunkowym, a nie we wnętrzu pętli. Taka konstrukcja została zaprezentowana na listingu 2.26. Jak widać, nie było to bardzo skomplikowane zadanie. Zwróćmy jednak uwagę, że nie uzyskaliśmy w ten sposób w pełni funkcjonalnego odpowiednika kodu z listingu 2.25. Co prawda, napisy zostaną wyświetcone, tak jak w poprzednim przypadku, 10 razy, ale wartość zmiennej wewnętrz pętli zmieniać się będzie od 1 do 10, a nie od 0 do 9! Wystarczy dodać do kodu część wyświetlającą stan zmiennej *i*, aby przekonać się, że tak jest faktycznie (widać to na rysunku 2.10). Byłaby to bardzo poważna zmiana, gdyby zmieniona *i* była wykorzystywana do konkretnych czynności, np. do indeksowania tablicy. Takimi zagadnieniami zajmiemy się jednak w dalszej części książki.

Listing 2.26. Modyfikacja zmiennej iteracyjnej w wyrażeniu warunkowym

```
var str = "";
i = 0;
while(i++ < 10){
    str += "Ten napis się powtarza... ";
    str += "<br />";
}
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

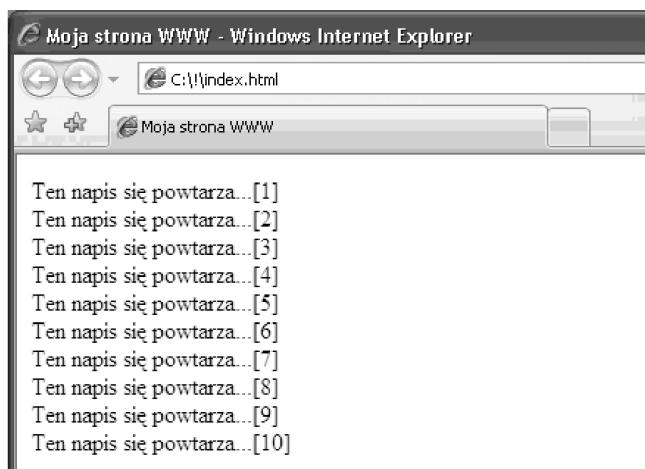
Pętla do...while

Kolejny rodzaj pętli występującej w JavaScriptie to *do...while*. Jak łatwo się domyślić, jest blisko spokrewniona ze zwykłą pętlą *while*. Oto jej ogólna postać:

```
do{
    instrukcje;
}
while(warunek);
```

Rysunek 2.10.

Modyfikacja skryptu spowodowała zmiany stanów zmiennej iteracyjnej



Oznacza to: „Wykonuj *instrukcje*, dopóki *warunek* jest prawdziwy”. A zatem, skoro tak dobrze szło nam do tej pory wyświetlanie dziesięciu napisów, zobaczymy, jak to zrobić za pomocą tego rodzaju pętli. Przykład jest widoczny na listingu 2.27.

Listing 2.27. Użycie pętli do...while

```
var str = "";
i = 0;
do{
    str += "Ten napis się powtarza...[" + i + "]";
    str += "<br />";
}
while(i++ < 9)

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Łatwo się domyślić, jaka jest podstawowa różnica w stosunku do pętli while. Otóż, w pętli while najpierw jest sprawdzany warunek, a dopiero potem wykonywane są instrukcje. W pętli do...while jest dokładnie odwrotnie — najpierw są wykonywane instrukcje, a dopiero potem sprawdzany jest warunek. Z tego też względu warunek jest nieco inny niż w poprzednim przykładzie. Tym razem sprawdzamy, czy *i* jest mniejsze od 9. Gdybyśmy pozostawili starą postać wyrażenia warunkowego w postaci *i++ < 10*, napis zostałby wyświetlony jedenaście razy!

Można temu zapobiec, wprowadzając wyrażenie modyfikujące zmienną *i* do wnętrza pętli; zatem sama pętla miałaby wtedy postać:

```
i = 0;
do{
    str += "Ten napis się powtarza...[" + i + "]";
    str += "<br />";
    i++;
}
while(i < 10)
```

Teraz warunek pozostaje w starej postaci i otrzymujemy odpowiednik pętli `while`.

Ta cecha (czyli wykonywanie instrukcji przed sprawdzeniem warunku) pętli `while` jest bardzo ważna, oznacza bowiem, że pętla tego typu jest wykonywana zawsze co najmniej raz nawet wtedy, jeśli warunek jest fałszywy. Można się o tym przekonać w bardzo prosty sposób — wprowadzając fałszywy warunek i obserwując zachowanie skryptu. Oto przykład:

```
i = 0;  
do{  
    str += "Ten napis się powtarza...[" + i + "]";  
    str += "<br />";  
    i++;  
}  
while(i < 0)
```

Warunek w tej postaci jest ewidentnie fałszywy, jako że zmienna `i` już w trakcie inicjalizacji jest równa 0 (więc nie może być jednocześnie mniejsza od 0!). Mimo to, po wykonaniu powyższego kodu na ekranie pojawi się jeden napis Ten napis się powtarza... Jest to najlepszy dowód na to, że warunek jest sprawdzany nie przed, ale po każdym przebiegu pętli.

Pętla `for...in`

Pętla typu `for...in` pozwala na odczytanie wartości oraz nazw właściwości obiektów (w tym również tablic). Ma schematyczną postać:

```
for (var nazwa in obiekt){  
    //instrukcje  
}
```

W takim przypadku we wnętrzu pętli pod identyfikator `nazwa` podstawiane są kolejne właściwości obiektu `obiekt`. Przykład jej użycia zostanie podany w dalszej części książki.

Jak nazywać zmienne iteracyjne?

Nazwy zmiennych iteracyjnych z formalnego punktu widzenia mogą być dowolne, o ile tylko spełniają ogólne zasady nazewnictwa tych konstrukcji programistycznych. Niemniej jednak zwyczajowo przyjmuje się, że są jednoliterowe i zaczynają się od litery `i`. Kolejne litery alfabetu (`j`, `k` itd.) są natomiast wykorzystywane w sytuacji, kiedy pojawiają się pętle zagnieżdżone, którymi zajmiemy się w dalszej części lekcji, lub też wtedy, gdy w jednej pętli używa się więcej niż jednej zmiennej iteracyjnej (takimi przypadkami nie będziemy się zajmować).

Zagnieżdżanie pętli

Wewnątrz każdej pętli można umieścić dowolne instrukcje, a więc i kolejną pętlę. Oznacza to, że mogą być one zagnieżdżane. Najczęściej zagnieżdżane są pętle for. Taka konstrukcja ma następującą postać:

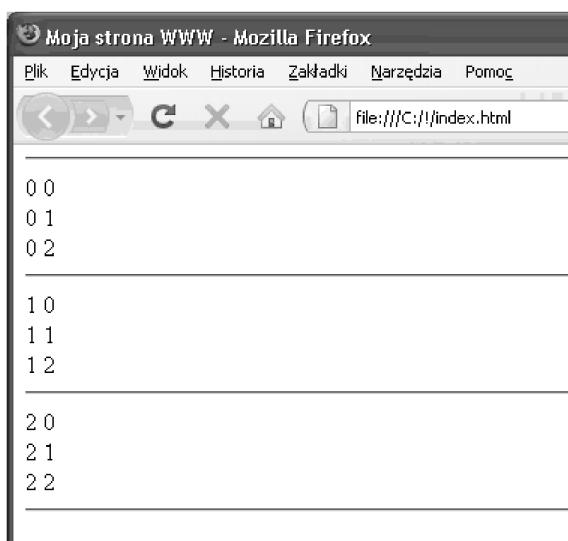
```
for(wyrażenie początkowe 1; wyrażenie warunkowe 1; wyrażenie modyfikujące 1){  
    for(wyrażenie początkowe 2; wyrażenie warunkowe 2; wyrażenie modyfikujące 2){  
        instrukcje do wykonania  
    }  
}
```

W takiej sytuacji w każdym przebiegu pętli zewnętrznej są wykonywane wszystkie przebiegi wewnętrznej. Zobaczmy to na przykładzie, który został przedstawiony na listingu 2.28, a efekt jego działania zobrazowano na rysunku 2.11.

Listing 2.28. Zagnieżdżone pętle for

```
var str = "<hr />";  
  
for(i = 0; i < 3; i++){  
    for(j = 0; j < 3; j++){  
        str += i + " " + j;  
        str += "<br />";  
    }  
    str += "<hr />";  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Rysunek 2.11.
Efekt działania
zagnieżdżonych
pętli for



W pętli zewnętrznej zmienną iteracyjną jest *i*, natomiast w wewnętrznej — *j*. Jest to zgodne ze przedstawionym wyżej schematem nazewnictwa. W pętli wewnętrznej znajduje się instrukcja:

```
str += i + " " + j;
```

dodająca stan tych zmiennych do zmiennej str, co ostatecznie spowoduje pojawienie się tych danych na ekranie. W pierwszej iteracji zewnętrznej *i* jest stała i równa 0, a *j* zmienia się od 0 do 2. W drugiej iteracji zewnętrznej *i* jest stała i równa 1, a *j* zmienia się od 0 do 2. Natomiast w trzeciej iteracji zewnętrznej *i* jest stała i równa 2, a *j* zmienia się od 0 do 2. Na rysunku 2.11 pierwsza kolumna obrazuje kolejne stany zmiennej *i*, a druga kolumna — *j*. Wartości generowane w kolejnych iteracjach zewnętrznych są oddzielone od siebie liniami poziomymi generowanymi za pomocą znacznika <hr />.

W podobny sposób można zagnieździć pętle while, choć tego typu konstrukcje są rzadziej spotykane. Istnieje również możliwość mieszania typów zagnieżdżonych pętli, tzn. w pętli while może być zagnieżdżona pętla for i odwrotnie — w pętli for może być zagnieżdżona pętla while. Jeżeli chcemy zagnieździć pętle while, możemy zrobić to np. w sposób przedstawiony na listingu 2.29.

Listing 2.29. Zagnieżdzone pętle while

```
var str = "<hr />";

i = 0;
while(i++ < 3){
    j = 0;
    while(j++ < 3){
        str += i + " " + j;
        str += "<br />";
    }
    str += "<hr />";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Ten skrypt działa analogicznie do przedstawionego na listingu 2.28, choć nie jest jego funkcjonalnym odpowiednikiem. W tym przykładzie bowiem inaczej zmienia się stan zmiennych iteracyjnych. Otóż, we wnętrzach obu pętli zmienne iteracyjne (*i* i *j*) zmieniają się od 1 do 3. Tak więc na ekranie pojawią się liczby o takim zakresie (a nie, jak w poprzednim przypadku, od 0 do 2). Co prawda, początkowym stanem *i* i *j* jest 0, ale wartość ta ulega zwiększeniu o 1 (za co odpowiada operator inkrementacji `++`) jeszcze przed rozpoczęciem wykonywania instrukcji wewnętrz pętli.

Zwróćmy też szczególną uwagę na umiejscowienie instrukcji inicjalizujących zmienne iteracyjne: *i* = 0; i *j* = 0;, a szczególnie na drugą z nich. Muszą one wystąpić bezpośrednio przed pętlami, których dotyczą, tzn. instrukcja *i* = 0; przed pętlą zewnętrzną, a instrukcja *j* = 0; przed pętlą wewnętrzną. W pętli for przypisanie tych wartości odbywało się w wyrażeniu początkowym. Warto się zastanowić, jaki byłby efekt, i dlaczego, gdyby instrukcja inicjalizująca zmienną *j* znalazła się przed obu pętlami, np.:

```
i = 0;
j = 0;
while(i++ < 3){
    while(j++ < 3){
        // tutaj dalsze instrukcje
    }
}
```

Przerywanie pętli

Wykonywanie pętli może zostać przerwane za pomocą instrukcji `break` (w języku angielskim `break` znaczy właśnie „przerywać”). Może ona zostać umieszczona w dowolnym miejscu w bloku pętli (pomiędzy znakami nawiasów klamrowego). Najczęściej używa się jej w połączeniu z instrukcją warunkową `if`. Instrukcję `break` poznaliśmy już wcześniej, przy omawianiu instrukcji wyboru `switch`. Tam też powodowała przerwanie działania i opuszczenie bloku `switch`. Przykład zastosowania `break` w pętli `while` został zaprezentowany na listingu 2.30.

Listing 2.30. Użycie instrukcji `break` do przerwania pętli

```
var str = "";

i = 0;
while(true){
    str += "Wartość zmiennej i to " + i;
    str += "<br />";
    if(i >= 9) break;
    i++;
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

W kodzie została umieszczona specyficzna pętla `while`, która powoduje, jakżeby inaczej, wyświetlenie dziesięciu napisów. Zwróćmy jednak uwagę na warunek kontynuowania pętli — jest to `true`, a zatem jest zawsze prawdziwy. Oczywiście, taka pętla wykonywałaby się w nieskończoność, gdyby nie instrukcja `break` znajdująca się wewnątrz. Z kolei warunkiem wykonania instrukcji `break` jest `i >= 9`, czyli przerwanie pętli następuje wtedy, kiedy zmieniona `i` osiągnie wartość 9 lub większą. Ponieważ `i` jest zwiększane o jeden w każdym przebiegu pętli (`i++`), a jego początkowa wartość to 0, mamy pewność, że pętla zostanie wykonana 10 razy, po czym będzie przerwana.

Instrukcję warunkową można też bez problemów połączyć z instrukcją zwiększającą wartość zmiennej. Wtedy warunek miałby postać:

```
if(i++ >= 9) break;
```

a funkcjonalność skryptu byłaby identyczna.

Spójrzmy również dokładniej na wyrażenie warunkowe, w którym występuje operator relacyjny `>=`. Instrukcja `break` faktycznie jest więc wykonywana, kiedy `i` będzie równe lub większe od 9. Czemu jednak nie użyliśmy zwykłego operatora `==`? Wszak instrukcja warunkowa w postaci:

```
if(i == 9) break;
```

też byłaby prawidłowa. Odpowiedź w pierwszej chwili zapewne wyda się dziwna — brzmi bowiem: „Na wszelki wypadek”. Nie jest to żart. Oczywiście, w przypadku kodu z listingu 2.30 wartość i nigdy nie przekroczy 9, a zatem nigdy też nie będzie prawdziwy warunek $i > 9$. W praktyce programistycznej warto jednak brać pod uwagę sytuacje, które podczas pisania kodu wydają się niemożliwe do spełnienia. Sprawdzi się to, kiedy będziemy tworzyć dużo bardziej skomplikowane kody. Nawet jednak na tym prostym przykładzie można pokazać, że lepiej było zastosować warunek $>=$ niż $==$. Wyobraźmy sobie bowiem, że pierwotnie napisaliśmy pętlę:

```
i = 0;
while(true){
    str += "Wartość zmiennej i to " + i;
    str += "<br />";
    if(i == 9) break;
    i++;
}
```

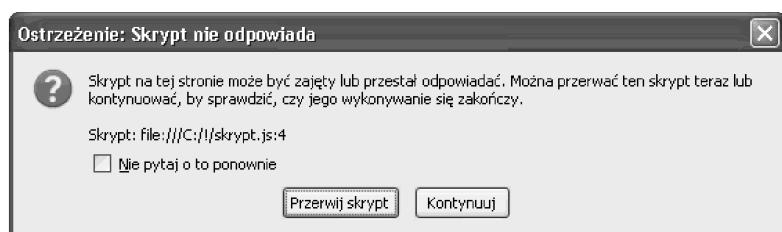
Po pewnym czasie jednak dokonaliśmy pewnej drobnej modyfikacji. Otóż, zaczęliśmy w każdym przebiegu zwiększać zmienną i nie o jeden, ale o dwa, nie korygując jednocześnie, nawet przez zwykłe zapomnienie, warunku zakończenia pętli. Powstał więc następujący kod:

```
i = 0;
while(true){
    str += "Wartość zmiennej i to " + i;
    str += "<br />";
    if(i == 9) break;
    i = i + 2;
}
```

Co się stało? Otóż to, czego za wszelką cenę musimy unikać. Napisaliśmy nieskończoną pętlę! Witryna z takim skryptem przestanie reagować na działania użytkownika. Na szczęście, żadna współczesna przeglądarka nie zawiesi się, ale wykryje, że skrypt zajmuje zbyt wiele zasobów systemowych i zaproponuje jego przerwanie. Przykładowe okno dialogowe z takim komunikatem widoczne jest na rysunku 2.12. Na pewno jednak nie jest to efekt, na jakim nam zależało.

Rysunek 2.12.

Przeglądarka zaproponowała przerwanie skryptu



Dlaczego taka pętla jest nieskończona? Otóż dlatego, że tym razem w każdym jej przebiegu zwiększamy wartość zmiennej i o 2 ($i = i + 2;$). Ponieważ początkową wartością i było 0, zmienna ta będzie przyjmowała kolejne wartości: 0, 2, 4, 6, 8, 10 itd. Jak widać, wartość 9 nigdy nie zostanie osiągnięta, a zatem warunek $i == 9$ zawsze będzie fałszywy. Tym samym nigdy nie zostanie wykonana instrukcja `break`.

Użycie operatora `>=` zapobiegłoby powstaniu takiego problemu. Skrypt być może nie działałby wtedy (po wstawieniu instrukcji `i = i + 2;`) w pełni zgodnie z założeniami, gdyż zapewne wymagałby również odpowiedniej zmiany warunku, jednak nie doprowadziłibyśmy do tak poważnej awarii, jak w prezentowanym przypadku.

Kontynuowanie pętli

Oprócz instrukcji `break`, która powodowała przerwanie wykonywania pętli oraz jej opuszczenie, istnieje instrukcja `continue` powodująca przejście do jej kolejnej iteracji, czyli jeśli wewnętrz pętli znajdzie się instrukcja `continue`, bieżąca iteracja (przebieg) zostanie przerwana oraz rozpoczęcie się kolejna (chyba że bieżąca iteracja była ostatnia). Zobaczmy, jak to działa, na konkretnym przykładzie. Na listingu 2.31 jest widoczna pętla `for`, która wyświetla liczby całkowite z zakresu 1 – 20 podzielne przez 2.

Listing 2.31. Użycie instrukcji `continue`

```
var str = "";

for(i = 1; i <= 20; i++){
    if(i % 2 != 0) continue;
    str += i;
    str += " ";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Pętla jest skonstruowana w dobrze już znany sposób. W środku znajduje się jednak instrukcja warunkowa badająca warunek `i % 2 != 0`, czyli sprawdzającą, czy reszta z dzielenia `i` przez 2 jest równa 0. Jeśli ten warunek jest fałszywy (reszta jest różna od 0), oznacza to, że wartość zapisana w `i` nie jest podzielna przez 2, jest zatem wykonywana instrukcja `continue`. Jak już wiemy, powoduje ona rozpoczęcie kolejnej iteracji pętli, tzn. zwiększenie wartości zmiennej `i` o jeden i przejście na początek pętli (do pierwszej instrukcji). Tym samym, jeśli wartość `i` jest niepodzielna przez dwa, nie zostanie wykonana znajdująca się za warunkiem instrukcja `str += i;`, co spowoduje, że dana wartość nie pojawi się na ekranie. Rzecz jasna, zadanie to można wykonać bez użycia instrukcji `continue` (ćwiczenia na końcu lekcji), ale bardzo dobrze ilustruje istotę jej działania. Efekt działania skryptu został zaprezentowany na rysunku 2.13.

Rysunek 2.13.

Wynik działania pętli
wyświetlającej
parzyste liczby
z zakresu 1 – 20



Warianty pętli for

Składnikiem pętli for są trzy wyrażenia: początkowe, warunkowe i modyfikujące. Okazuje się jednak, że można je umieszczać w dość dowolnych miejscach, a w przypadku ekstremalnym można wszystkie trzy wręcz wyrzucić z nawiasu okrągłego znajdującego się za słowem for. Sprawdźmy więc, jak elastyczny jest język JavaScript⁸.

Najprościej usunąć wyrażenie początkowe. Przenosimy je po prostu przed pętlę. Schemat takiego postępowania jest następujący:

```
wyrażenie początkowe
for (; wyrażenie modyfikujące; wyrażenie warunkowe;){
    instrukcje do wykonania
}
```

Jeżeli w skrypcie z listingu 2.24 chcielibyśmy pozbyć się z pętli wyrażenia początkowego, przyjęłaby ona postać:

```
i = 0;
for(; i < 10; i++){
    str += "Ten napis się powtarza...";
    str += "<br />";
}
```

Efekt działania będzie identyczny. Zwróciły jednak uwagę na średniki w nawiasie okrągłym pętli. Mimo iż wyrażenie początkowe znalazło się teraz przed pętlą, pierwszy średnik nadal jest niezbędny. Jeśli go zabraknie, skrypt z pewnością nie zadziała.

Pozostawmy teraz wyrażenie początkowe, ale usuńmy modyfikujące. Oczywiście, nie możemy pozbyć się go całkowicie, ale możemy przenieść je do wnętrza pętli. Schematycznie będzie to wyglądało tak:

```
for (wyrażenie początkowe; wyrażenie warunkowe;){
    instrukcje do wykonania
    wyrażenie modyfikujące;
}
```

Jeśli zastosujemy ten schemat do pętli z listingu 2.24, otrzymamy kod:

```
for(i = 0; i < 10;){
    str += "Ten napis się powtarza...";
    str += "<br />";
    i++;
}
```

To nadal odpowiednik pierwszego przykładu i działa w identyczny sposób. Tu też nie możemy zapomnieć o średnikach za wyrażeniami. Jeden musi wystąpić po wyrażeniu początkowym, a drugi — po warunkowym.

⁸ Równie elastyczne są inne języki programowania oparte na podobnej koncepcji składni. Takie same efekty można też osiągnąć w C, C++, Java, PHP itp.

Można pozbyć się jednocześnie wyrażeń początkowego i modyfikującego. Schematycznie zrobimy to tak:

```
wyrażenie początkowe:  
for (: wyrażenie warunkowe;){  
    instrukcje do wykonania  
    wyrażenie modyfikujące;  
}
```

a w praktyce:

```
i = 0;  
for(: i < 10;){  
    str += "Ten napis się powtarza... [" + i + "]";  
    str += "<br />";  
    i++;  
}
```

W tym przykładzie w nawiasie okrągły występującym po for jest już tylko jedno wyrażenie, ale ponownie trzeba tu zwrócić uwagę na średniki. Pierwszy kończy wyrażenie początkowe (mimo że w tym miejscu jest ono puste!), drugi — wyrażenie warunkowe.

Takich kombinacji można wymyślić jeszcze kilka. Nie będziemy wszystkich przytaczać. Na zakończenie wypróbujmy więc jeszcze jedną. Z nawiasu okragłego można usunąć wszystkie wyrażenia, a pętla zachowa pełną funkcjonalność. Jak to zrobić? Wyrażenie początkowe trzeba przenieść przed pętlę, a modyfikujące i warunkowe do wnętrza pętli:

```
wyrażenie początkowe:  
for (::){  
    instrukcje do wykonania  
    wyrażenie modyfikujące  
    wyrażenie warunkowe i instrukcja przerywająca działanie pętli  
}
```

I tu ponownie, ale już ostatni raz, zwrócić uwagę na średniki. Mimo braku wszystkich wyrażeń w nawiasie okrągły i nieco dziwnego wyglądu takiej konstrukcji, oba te znaki są konieczne do prawidłowego działania pętli. W praktyce nasza pętla mogłaby przyjąć postać:

```
i = 0;  
for(::){  
    str += "Ten napis się powtarza... [" + i + "]";  
    str += "<br />";  
    i++;  
    if(i > 9) break;  
}
```

Widzimy, że wyrażenie początkowe zostało przesunięte przed pętlę, natomiast warunkowe i modyfikujące do jej wnętrza. Niezbędne było również użycie instrukcji break, pozwala ona bowiem na przerwanie pętli. Inny jest także warunek zakończenia, tym razem jest to $i > 9$, a więc pętla zostanie zakończona za pomocą instrukcji break, gdy zmienna i przekroczy wartość 9. Warunek ten jest, można powiedzieć, odwrócony w stosunku do poprzednich rozwiązań. Dzieje się tak dlatego, że poprzednio służył

do zbadania, jak długo pętla ma być kontynuowana (kontynuuj działanie pętli dopóty, dopóki $i < 10$), a tym razem służy do zbadania, kiedy pętla ma zostać zakończona (przerwij działanie pętli, gdy $i > 9$). Ta ostatnia pętla, mimo swej nietypowej postaci, to jednak wciąż w pełni funkcjonalny odpowiednik pętli zaprezentowanej na listingu 2.24.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 5.1.

Zmodyfikuj kod z listingu 2.26 w taki sposób, aby inkrementacja zmiennej i odbywała się w wyrażeniu warunkowym pętli, a skrypt był funkcjonalnym odpowiednikiem tego z listingu 2.25.

Ćwiczenie 5.2.

Zmień kod skryptu z listingu 2.29, tak aby wynik działania był taki sam jak w przypadku kodu z listingu 2.28. Wykonaj dwie wersje ćwiczenia. W pierwszej pozostaw bez zmian występujące w pętlach wyrażenia warunkowe $i++ < 3$ i $j++ < 3$. W drugiej nie używaj operatorów inkrementacji w wyrażeniach warunkowych pętli.

Ćwiczenie 5.3.

Zmień kod z listingu 2.31, tak aby nie zmienił się sposób działania skryptu (aby były wyświetlane liczby z zakresu 1 – 20 podzielne przez dwa), ale nie było konieczności używania instrukcji `continue`.

Ćwiczenie 5.4.

Napisz skrypt, który za pomocą pętli `while` i instrukcji `continue` wyświetli nieparzyste liczby z zakresu 10 – 30, pominie jednak wartości 15, 21 i 27.

Ćwiczenie 5.5.

Napisz skrypt będący funkcjonalnym odpowiednikiem tego z listingu 2.24, w którym w pętli `for` z nawiasu okrągłego zostało usunięte wyrażenie warunkowe.

Lekcja 6. Funkcje i zasięg zmiennych

Funkcje to wydzielone fragmenty kodu służące do wykonywania konkretnych zadań. Im właśnie poświęcona jest cała lekcja 6. Dowiemy się, jak definiować funkcje i jak ich używać oraz czemu ułatwiają pracę programistom. Poznamy sposoby przekazywania danych do funkcji, czyli ich argumenty, a także zobaczymy, jak funkcje mogą zwracać dane, np. wyniki wykonanych przez nie obliczeń. Nie pominiemy także ważnego tematu dotyczącego tzw. widoczności, czyli zasięgu zmiennych.

Jak tworzymy funkcje?

Funkcje tworzy się za pomocą słowa `function`. Ogólna definicja ma postać:

```
function nazwa_funkcji()
{
    //instrukcje wnętrza funkcji
}
```

Przy nazywaniu funkcji obowiązują zasady, podobne jak przy zmiennych (a ogólniej — identyfikatorach, lekcja 2.). Nazwa może się składać z liter, cyfr oraz znaków podkreślenia i dolara, nie może jednak zaczynać się od cyfry. We wszystkich współczesnych implementacjach JavaScriptu może też zawierać znaki spoza alfabetu łacińskiego (np. polskie znaki diakrytyczne), choć z reguły się ich nie stosuje.

Aby wykonać instrukcje znajdujące się wewnętrz funkcji (pomiędzy znakami nawiasów klamrowych), należy ją wywołać. Wywołanie polega na umieszczeniu w kodzie skryptu nazwy funkcji wraz z występującym po niej nawiasem okrągłym. Utwórzmy możliwie najprostszą funkcję i sprawdźmy, jak coś takiego działa w praktyce. Odpowiedni przykład został zaprezentowany na listingu 2.32.

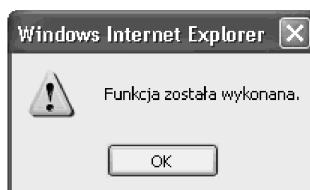
Listing 2.32. Użycie prostej funkcji

```
function wyswietl()
{
    alert("Funkcja została wykonana.");
}

wyswietl();
```

W skrypcie najpierw zdefiniowano funkcję o nazwie `wyswietl`, która następnie została wywołana przez podanie jej nazwy zakończonej znakami nawiasów okrągłego. Wewnątrz funkcji (stosuje się też termin: w ciele funkcji) umieszczona jest instrukcja `alert`. Jak wiemy z wcześniejszych lekcji, powoduje ona wyświetlenie okna dialogowego. Ponieważ wywołanie funkcji jest równoznaczne z wykonaniem znajdujących się w niej instrukcji, w przeglądarce zobaczymy zdefiniowany napis (rysunek 2.14).

Rysunek 2.14.
Potwierdzenie
wykonania funkcji



Po co jednak tworzyć funkcje? Po pierwsze, po to, aby niepotrzebnie nie powtarzać wspólnych elementów kodu. Jeśli np. w skrypcie w różnych jego miejscach wielokrotnie potrzebowalibyśmy wykonywać pewien zestaw instrukcji, zamiast wielokrotnie je wpisywać, lepiej zebrać je w jednym miejscu, tworząc funkcję. Po drugie, aby podzielić skrypt na mniejsze, funkcjonalne fragmenty, którymi wtedy o wiele łatwiej zarządzać.

Argumenty funkcji

Funkcjom można przekazywać argumenty, czyli wartości (dane), które mogą wpływać na ich zachowanie lub też być przez nie przetwarzane. Listę argumentów oddzielonych od siebie przecinkami należy umieścić w nawiasie okrągłym za nazwą funkcji. A zatem taka konstrukcja ma schematyczną postać:

```
function nazwa_funkcji(argument1, argument2, ..., argumentN)
{
    //instrukcje wnętrza funkcji
}
```

Napiszemy funkcję, której zadaniem będzie wyświetlenie wartości przekazanego jej argumentu, i wywołamy ją w kodzie skryptu. Działający w taki sposób kod został zaprezentowany na listingu 2.33.

Listing 2.33. Przekazanie funkcji argumentu

```
function wyswietl(napis)
{
    alert(napis);
}

wyswietl("To jest napis.");
```

Powstała tu funkcja `wyswietl`, która przyjmuje jeden argument o nazwie `napis`. Wewnątrz funkcji można odczytać wartość tego argumentu, odwołując się do jego nazwy. Tym samym w instrukcji:

```
alert(napis);
```

argument `napis` zostanie zamieniony na ciąg znaków `To jest napis.`, co powoduje wyświetlenie tego ciągu na ekranie. Warto wspomnieć, że w ten sposób można przekazać argument dowolnego typu, typ nie jest bowiem z góry określony. Prawidłowe będą więc np. wywołania:

```
wyswietl("To jest napis.");
wyswietl(128);
wyswietl(3.14);
wyswietl(true);
```

Mogą one występować jedno za drugim. Jeśli zastosujemy kod widoczny na listingu 2.34, na ekranie pojawią się pod rząd cztery okna dialogowe z różnymi wartościami.

Listing 2.34. Wielokrotne wywoływanie funkcji

```
function wyswietl(napis)
{
    alert(napis);
}

wyswietl("To jest napis.");
wyswietl(128);
wyswietl(3.14);
wyswietl(true);
```

Zgodnie z podanym wyżej schematem, liczba argumentów funkcji nie jest z góry ograniczona — może być ich dowolnie dużo. W praktyce najczęściej występują funkcje o liczbie argumentów nieprzekraczających 3 – 4, ale, choć dużo rzadziej, spotkamy się też z takimi, które przyjmują 7 czy 8. Przy użyciu argumentów możemy przekazać funkcji różne dane, które ta może dowolnie przetwarzanie i wykonywać na nich różne operacje. Wykonajmy prosty przykład. Niech powstanie funkcja o nazwie dodaj, przyjmująca dwa argumenty, która wyświetli wynik ich dodawania. Działający w taki sposób kod został zaprezentowany na listingu 2.35.

Listing 2.35. Funkcja wykonująca operacje na argumentach

```
function dodaj(argument1, argument2)
{
    var suma = argument1 + argument2;
    var komunikat = argument1 + " + " + argument2;
    komunikat += " = " + suma;
    alert(komunikat);
}

dodaj(15, 22);
```

Funkcja dodaj przyjmuje dwa argumenty o nazwach argument1 i argument2. Pierwsza instrukcja we wnętrzu tej funkcji powoduje utworzenie pomocniczej zmiennej suma i przypisanie jej wyniku dodawania argumentów:

```
var suma = argument1 + argument2;
```

Następnie tworzona jest druga zmienna pomocnicza — komunikat. Jest jej przypisywany ciąg znaków obrazujący wykonywanie działania, czyli kolejno: wartość pierwszego argumentu, znak dodawania, wartość drugiego argumentu, znak równości, wartość zmiennej suma. Zmienna komunikat jest używana w instrukcji alert do wyświetlenia komunikatu zawierającego równanie.

Ponieważ w dalszej części kodu funkcja dodaj została wywołana z argumentami 15 i 22:

```
dodaj(15, 22);
```

po uruchomieniu skryptu na ekranie pojawi się okno dialogowe widoczne na rysunku 2.15.

Rysunek 2.15.
Wynik działania
funkcji dodającej
wartości argumentów



Zwracanie wartości przez funkcję

Przyjmowanie argumentów to nie jedyna cecha funkcji — mogą one również zwracać różne wartości. Czynność ta jest wykonywana za pomocą instrukcji return. Jeśli wystąpi ona wewnętrz funkcji, ta jest przerywana i zwraca wartość występującą po return. Schematycznie tego typu konstrukcja wygląda następująco:

```
function nazwa_funkcji(argumenty)
{
    //instrukcje wewnętrza funkcji
    return wartość;
}
```

Jeśli wywołamy taką funkcję, w miejscu jej wywołania zostanie wstawiona zwrócona przez nią wartość, która będzie mogła być wykorzystana w dalszej części skryptu. Warto też wiedzieć, że użycie samej instrukcji return bez żadnych argumentów również powoduje przerwanie działania funkcji.

Żeby zobaczyć, jak opisana technika działa w praktyce, napiszemy funkcję, której zadanie będzie podobne do zadania ze skryptu z listingu 3.35. Będzie przyjmowała dwa argumenty i dodawała je do siebie. Różnica będzie taka, że tym razem wynik dodawania nie będzie przez funkcję wyświetlany, ale zwracany do miejsca wywołania. Kod działający w taki sposób został zaprezentowany na listingu 2.36.

Listing 2.36. Funkcja zwracająca wartość

```
function dodaj(argument1, argument2)
{
    var suma = argument1 + argument2;
    return suma;
}

var wynik = dodaj(15, 22);
var str = "Wynikiem dodawania 15 + 22 jest wartość ";
str += wynik + ":";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Funkcja dodaj przyjmuje argumenty argument1 i argument2. W jej wnętrzu są one dodawane za pomocą arytmetycznego operatora +, a wynik tego działania jest przypisywany zmiennej pomocniczej suma. Wartość tej zmiennej jest zwracana za pomocą instrukcji return. W dalszej części kodu funkcja dodaj jest wywoływana z argumentami 15 i 22, a wynik jej działania (czyli zwrócona przez nią wartość) jest przypisywany zmiennej wynik, która następnie używana jest do skonstruowania (na takiej samej zasadzie jak we wcześniejszych przykładach) komunikatu, jaki ma się pojawić na ekranie. Wynik działania skryptu został zaprezentowany na rysunku 2.16.

Rysunek 2.16.
Wynik działania skryptu używającego funkcji do dodania dwóch liczb



Instrukcja

```
var wynik = dodaj(15, 22);
```

działa następująco.

- ◆ Najpierw wywoływana jest funkcja dodaj, a zatem są wykonywane jej instrukcje.
- ◆ Następnie wynik działania funkcji dodaj jest podstawiany w miejscu jej wywołania. Ponieważ w tym przypadku jest to 37, instrukcja jest traktowana jako wynik = 37;.
- ◆ Zmiennej wynik jest przypisywana wartość zwrócona przez funkcję.

Przekonajmy się jeszcze, że pusta instrukcja return też może zostać użyta i powoduje przerwanie działania funkcji. Spójrzmy na listing 2.37.

Listing 2.37. Pusta instrukcja return

```
function wyswietl()
{
    alert("Pierwszy komunikat");
    return;
    alert("Drugi komunikat");
}

wyswietl();
```

Zdefiniowana została funkcja wyswietl zawierająca dwie instrukcje alert, pomiędzy którymi występuje instrukcja return. Gdyby instrukcji return nie było, po uruchomieniu skryptu na ekranie pojawiłyby się następujące po sobie dwa okna dialogowe z komunikatami. Ponieważ jednak instrukcja return występuje i przerywa działanie funkcji, to zgodnie z tym, co zostało napisane wyżej, druga instrukcja alert nie zostanie wykonana, o czym można się przekonać, uruchamiając kod.

Zasięg zmiennych

Omówimy teraz temat zasięgu zmiennych (używa się również terminu *widoczność zmiennych*). Zasięg możemy określić jako miejsca, w których zmienna jest ważna i można się do niej bezpośrednio odwoływać. Zmienna może być:

- ◆ globalna,
- ◆ lokalna.

Zmienne globalne (o zasięgu globalnym) to takie, które są widoczne w każdym miejscu skryptu. Zmienna jest globalna, jeśli została zdefiniowana poza wnętrzami funkcji. Wtedy można się do niej odwoływać w dowolnym miejscu kodu (również we wnętrzach funkcji). Rozważmy przykład widoczny na listingu 2.38.

Listing 2.38. Dostęp do zmiennej globalnej

```
var liczba = 100;
var str = "";

function wyswietl()
{
    str += "Funkcja wyswietl: liczba = ";
    str += liczba;
    str += ".<br />";
    liczba = 200;
}

str += "Przed wywołaniem f. wyswietl liczba = ";
str += liczba;
str += ".<br />";

wyswietl();

str += "Po wywołaniu f. wyswietl liczba = ";
str += liczba;
str += ".<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

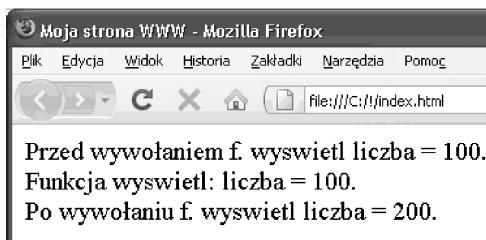
Na początku zostały zadeklarowane dwie zmienne: str i liczba. Obie są globalne, co oznacza, że można się do nich odwoływać w każdym miejscu skryptu. Dalej występuje definicja funkcji wyswietl — pominiemy ją na razie. Za funkcją znajdują się instrukcje przypisujące zmiennej str komunikat informujący o stanie zmiennej liczba. Oczywiście, ponieważ zmiennej tej została przypisana wartość 100, taka też wartość znajdzie się w komunikacie. Następnie wywoływana jest funkcja wyswietl. Najpierw odczytuje ona stan zmiennej liczba i dodaje do zmiennej str kolejny komunikat, a następnie przypisuje zmiennej liczba wartość 200. Widać więc wyraźnie, że we wnętrzu funkcji z powodzeniem mogliśmy odwołać się do obu zmiennych globalnych zarówno przy odczytcie, jak i zapisie.

Po wywołaniu funkcji wyswietl w głównej części skryptu ponownie konstruowany jest komunikat informujący o stanie zmiennej liczba. W ten sposób sprawdzamy, czy modyfikacja wartości zmiennej globalnej we wnętrzu funkcji obowiązuje po zakończeniu funkcji. Tak być powinno i tak jest. Zmienna liczba zmieniła swoją zawartość. Gdy uruchomimy skrypt, na ekranie zobaczymy komunikaty zaprezentowane na rysunku 2.17.

Inaczej jest ze zmiennymi lokalnymi. Zmienne lokalne to takie, które zostały zdefiniowane we wnętrzu funkcji z użyciem słowa var. Zmienna lokalna obowiązuje tylko i wyłącznie w funkcji, w której została umieszczona. Spoza funkcji nie można się do niej odwoływać. Sprawdźmy. Na listingu 2.39 widnieje kod skryptu, który próbuje wykonać błędne odwołanie.

Rysunek 2.17.

Komunikaty informujące o stanie zmiennej przed wywołaniem funkcji i po nim

**Listing 2.39.** Próba dostępu do zmiennej lokalnej

```
function wyswietl()
{
    var liczba = 200;
    alert(liczba);
}
wyswietl();
alert(liczba); //ta instrukcja jest błędna
alert("Koniec skryptu..");
```

W funkcji `wyswietl` zdefiniowano zmienną `liczba` i przypisano jej wartość 200. Wartość zmiennej jest też wyświetlana w oknie dialogowym za pomocą instrukcji `alert`. W kodzie następuje też wywołanie funkcji `wyswietl`, a więc wartość 200 pojawi się na ekranie. To nie powinno budzić naszych wątpliwości. Zdecydowanie nieprawidłowa jest jednak przedostatnia instrukcja skryptu, która próbuje odwołać się do zmiennej zdefiniowanej w funkcji poza tą funkcją. Takiej operacji nie można wykonać, zatem spowoduje przerwanie wykonywania kodu. Nie tylko nie zostanie wykonana, ale też cały skrypt zostanie zatrzymany. Nie zobaczymy więc również komunikatu kończącego, generowanego przez ostatnią instrukcję `alert`.

Zwróćmy jednak szczególną uwagę na podaną wyżej definicję zmiennej lokalnej. Musi ona spełniać dwa warunki.

1. Jej definicja musi znajdować się wewnątrz funkcji.
2. W definicji musi zostać użyte słowo `var`.

Bardzo ciekawa jest odpowiedź na pytanie, co się stanie, jeśli we wnętrzu funkcji utworzymy zmienną bez słowa `var`, np.:

```
function wyswietl()
{
    liczba = 200;
}
```

Otoż, w ten sposób powstanie zmienna globalna! Będzie dostępna w każdym miejscu skryptu, jeśli tylko zostanie wywołana zawierająca ją funkcja. Zobrazowano to w skrypcie z listingu 2.40.

Listing 2.40. Definicja zmiennej globalnej we wnętrzu funkcji

```
// poniższa instrukcja jest błędna
// alert(liczba);
function wyswietl()
{
    liczba = 200;
}
wyswietl();
// poniższa instrukcja jest prawidłowa
alert(liczba);
alert("Koniec skryptu.");
```

Pierwsza instrukcja alert została poprzedzona znakiem komentarza, gdyż jest błędna. W miejscu jej wystąpienia nie została jeszcze zdefiniowana zmienna o nazwie liczba, nie można się więc do niej odwoływać. Jednak po wywołaniu funkcji wyswietl można się już odwoływać do zmiennej liczba. Dzieje się tak dlatego, że zmienna ta została zadeklarowana bez użycia słowa var. Tym samym, druga instrukcja alert jest prawidłowa. Zostanie też wykonana trzecia instrukcja alert oznajmiająca koniec wykonywania skryptu.

Jak przekazywane są argumenty?

Waźną kwestią, na którą również musimy zwrócić uwagę, jest sposób przekazywania funkcjom argumentów. W klasycznych językach programowania (i niektórych skryptowych) zazwyczaj mamy do wyboru dwa sposoby: przez wartość (z ang. *by value*) i przez referencję (z ang. *by reference*). W uproszczeniu mówiąc, pierwszy sposób polega na tym, że funkcja operuje na kopii danych, a drugi — że operuje na danych oryginalnych. W JavaScripte argumenty są przekazywane tylko przez wartość. Na czym to polega, najlepiej pokazać na przykładzie. Można go zobaczyć na listingu 2.41.

Listing 2.41. Przekazywanie argumentów przez wartość

```
var liczba = 100;

function abc(parametr)
{
    parametr = 200;
    alert(parametr);
}

alert("liczba = " + liczba);
abc(liczba);
alert("liczba = " + liczba);
```

W skrypcie mamy zdefiniowaną zmienną liczba o wartości 100 oraz funkcję abc. Funkcja przyjmuje jeden argument o nazwie parametr. Zmienia jego wartość na 200 oraz wyświetla ją w oknie dialogowym za pomocą instrukcji alert. W dalszej części kodu najpierw wyświetlana jest aktualna wartość zmiennej liczba, następnie jest wywoływana funkcja abc i na zakończenie ponownie wyświetlany stan zmiennej liczba.

Zwróćmy uwagę, że jako argument funkcji została użyta zmienna `liczba`. Czy zatem jej wartość zostanie zmieniona na 200? Po uruchomieniu skryptu przekonamy się, że nie. Zarówno przed wywołaniem funkcji, jak i po jej wywołaniu wartością zmiennej będzie 100. Jedynie we wnętrzu funkcji `abc` argument przyjmie wartość 200. Dzieje się tak właśnie dlatego, że przekazywanie argumentów odbywa się przez wartość, tzn. że w instrukcji:

```
abc(liczba);
```

funkcji `abc` nie została przekazana oryginalna zmienna `liczba`, ale jej wartość czy też, dokładniej, kopia. Wszelkie operacje we wnętrzu były wykonywane na owej kopii, a nie na oryginalnej zmiennej.

Spójrzmy jeszcze na listing 2.42. Jak należy interpretować taki kod? Co jest modyfikowane w funkcji `abc` w wierszu:

```
liczba = 200;
```

Listing 2.42. Nazewnictwo zmiennych i argumentów

```
var liczba = 100;

function abc(liczba)
{
    liczba = 200;
    alert(liczba);
}

alert("liczba = " + liczba);
abc(liczba);
alert("liczba = " + liczba);
```

Po przeczytaniu części lekcji dotyczącej zasięgu moglibyśmy pomyśleć, że zmieniana jest wartość globalnej zmiennej `liczba`. Jednak to nie jest właściwy wniosek. Występuje tu nakładanie się nazw. Globalna zmienna `liczba` ma taką samą nazwę jak argument funkcji `abc`. W takiej sytuacji, pisząc we wnętrzu funkcji słowo `liczba`, zawsze odwołujemy się do argumentu, można powiedzieć, że przesyłania on wtedy zmienną globalną. Dostęp do tej zmiennej nadal jest możliwy, ale wymaga użycia dodatkowej konstrukcji programistycznej. Tym tematem zajmiemy się w rozdziale 3.

Pomijanie argumentów

Jeśli funkcja przyjmuje argumenty, należy je podać w trakcie jej wywołania. Skoro taka funkcja jak `dodaj` przyjmowała dwa argumenty, wywołyвалиśmy ją np. w sposób następujący:

```
dodaj(15, 22);
```

Czy jest to obligatoryjne postępowanie? Otóż nie. W wywołaniu można bowiem pominąć dowolną liczbę argumentów (również wszystkie), przy czym, co wydaje się oczywiste i logiczne, można je pomijać sukcesywnie od strony prawej do lewej. Znaczy to, że jeśli przyjmowane są dwa argumenty, można pominąć albo drugi, albo

i drugi i pierwszy; jeśli przyjmowane są trzy, można pominąć albo trzeci, albo trzeci i drugi, albo trzeci, drugi i pierwszy. Jaka jest zatem wartość takiego pominiętego argumentu? Jest to wartość specjalna undefined.

Załóżmy, że mamy funkcję abc, przyjmującą dwa argumenty i wyświetlającą ich wartości za pomocą instrukcji alert. Będzie ona miała postać:

```
function abc(argument1, argument2)
{
    var komunikat = "argument1 = " + argument1;
    komunikat += "\nargument2 = " + argument2;
    alert(komunikat);
}
```

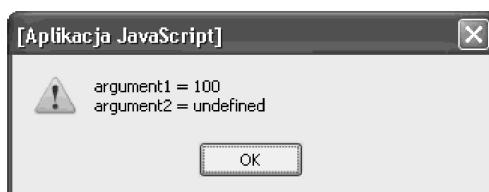
i zostanie trzykrotnie wywołana za pomocą następujących instrukcji:

```
abc(100, 200);
abc(100);
abc();
```

Pierwsze wywołanie jest jasne. Funkcja przyjmuje dwa argumenty i oba zostały podane. W drugim przypadku pominięty został drugi argument. Nie jest to błędem, przyjmie on po prostu wartość undefined, o czym przekona nas treść wyświetlanego przez funkcję okna dialogowego (rysunek 2.18). W trzecim przypadku funkcja została wywołana bez żadnych argumentów, co oznacza, że oba przyjmą wartość undefined.

Rysunek 2.18.

Pominięty argument
przyjmuje wartość
undefined



Czy ta właściwość jest wykorzystywana w praktyce? Oczywiście. Można to pokazać nawet na tak prostym przykładzie jak funkcja dodająca dwie wartości. Mogłaby ona przecież działać tak, że gdy otrzyma oba argumenty, doda je po prostu do siebie, kiedy jednak otrzyma tylko pierwszy, doda do niego z góry ustaloną wartość, np. 100. Tak działający kod został przedstawiony na listingu 2.43.

Listing 2.43. Wykorzystanie niezdefiniowanych argumentów

```
function dodaj(argument1, argument2)
{
    if(argument2 == undefined){
        argument2 = 100;
    }
    return argument1 + argument2;
}

var wynik1 = dodaj(15, 22);
var wynik2 = dodaj(18);
```

```

var str = "Wywołanie dodaj(15, 22) ";
str += "dało w wyniku wartość " + wynik1;
str += ".<br />Wywołanie dodaj(18) ";
str += "dało w wyniku wartość " + wynik2;
str += ".";

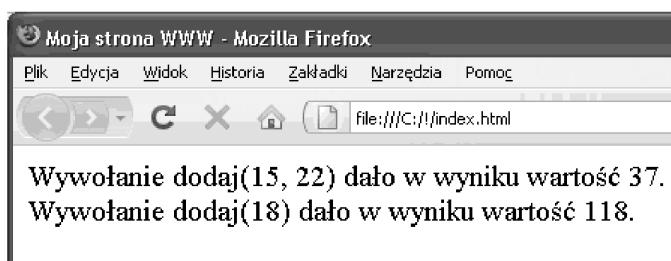
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Jak widać, definicja takiej funkcji jest bardzo prosta. Przyjmuje ona dwa argumenty: argument1 i argument2. We wnętrzu funkcji znajduje się instrukcja warunkowa if bieżąca, czy drugi z argumentów ma wartość undefined. Jeśli tak jest, przypisuje mu wartość 100. Jeżeli wartość argumentu jest różna od undefined, pozostaje niezmieniona. Na zakończenie kodu funkcji, za pomocą instrukcji return zwracany jest wynik dodawania argument1 + argument2.

W dalszej części kodu funkcja dodaj jest wywoływana dwukrotnie, raz z dwoma argumentami, a raz z jednym. Wyniki tych wywołań są przypisywane zmiennym wynik1 i wynik2. Zmienne są następnie używane przy tworzeniu komunikatu, który ma pojawić się na stronie. Komunikat jest przypisywany zmiennej str (zgodnie z podanym na początku rozdziału schematem). Ostatecznie po uruchomieniu skryptu ukaże się widok zaprezentowany na rysunku 2.19. Tak więc wywołanie funkcji dodaj bez drugiego argumentu zakończyło się pełnym powodzeniem.

Rysunek 2.19.
Wywołanie funkcji
dodaj z różnymi
argumentami



Dociekliwi czytelnicy zadadzą zapewne w tym miejscu dwa pytania. Po pierwsze, co się stanie, jeśli funkcja dodaj z listingu 2.43 zostanie wywołana bez żadnych argumentów. Czy nastąpi awaria skryptu, a jeśli nie, to jaki będzie wynik jego działania? Warto sprawdzić to samodzielnie, wprowadzając odpowiednie poprawki w kodzie.

Po drugie, czy można wywołać dowolną funkcję, podając więcej argumentów, niż zostało to zadeklarowane w jej definicji? Czy np. można wywołać funkcję dodaj z trzema argumentami? Odpowiedź jest twierdząca! Aby jednak skorzystać z takiego nadprogramowego argumentu, trzeba użyć dodatkowych konstrukcji programistycznych. Omówimy ten temat w rozdziale 3., w którym okaże się, że funkcje są obiektami.

Funkcje wewnętrzne

Funkcja może być zdefiniowana wewnętrz w innej funkcji — mówimy wtedy o funkcjach wewnętrznych. Takie konstrukcje nie są bardzo często używane i nie będziemy z nich korzystać w dalszej części kursu, trzeba jednak wiedzieć o ich istnieniu oraz znać interpretację. Definicja funkcji wewnętrz innej funkcji nie różni się niczym od klasycznej. Schematyczna konstrukcja jest następująca:

```
function funkcja_zewnętrzna(argumenty){  
    function funkcja_wewnętrzna(argumenty){  
        //instrukcje funkcji wewnętrznej  
    }  
    //instrukcje funkcji zewnętrznej  
}
```

Można też zastosować kilka funkcji wewnętrznych. Z funkcji wewnętrznej można korzystać tylko wewnętrz funkcji zewnętrznej. Dla pozostałej części skryptu jest niewidoczna. Przykład użycia funkcji wewnętrznej został przedstawiony na listingu 2.44.

Listing 2.44. Użycie funkcji wewnętrznej

```
function obliczenia(argument1, argument2)  
{  
    function kwadrat(argument){  
        return argument * argument;  
    }  
  
    return kwadrat(argument1) + kwadrat(argument2);  
}  
  
var wynik = obliczenia(2, 4);  
var str = "Suma kwadratów 2 i 4 to " + wynik;  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Mamy tu funkcję obliczenia przyjmującą dwa argumenty. Jej zadaniem jest zwrócenie sumy kwadratów tych argumentów, czyli wykonanie działania $argument1^2 + argument2^2$. Korzysta ona z pomocniczej funkcji wewnętrznej o nazwie kwadrat. Zadaniem tej funkcji jest zwrócenie wartości wynikającej z podniesienia argumentu do kwadratu (czyli do drugiej potęgi). To działanie jest wykonywane w najprostszym możliwy sposób, czyli przez pomnożenie argumentu przez siebie. Wynik jest zwracany za pomocą instrukcji return:

```
return argument * argument;
```

Funkcja obliczenia używa swojej wewnętrznej funkcji kwadrat w instrukcji:

```
return kwadrat(argument1) + kwadrat(argument2);
```

Najpierw zostanie więc wykonane działanie:

```
kwadrat(argument1) + kwadrat(argument2)
```

Następnie jego wynik zostanie zwrócony jako rezultat działania funkcji obliczenia. Wywołanie:

```
kwadrat(argument1)
```

zwraca argument argument1 podniesiony do drugiej potęgi, a:

```
kwadrat(argument2)
```

argument argument2 podniesiony do drugiej potęgi. Ostatecznym wynikiem będzie więc suma argumentów podniesionych do kwadratu.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 6.1.

Napisz funkcję, której dwa pierwsze argumenty będą określały operandy, a trzeci — rodzaj wykonywanego działania. W zależności od stanu trzeciego argumentu, funkcja ma zwrócić sumę, różnicę, iloczyn, iloraz lub resztę z dzielenia pierwszego i drugiego argumentu.

Ćwiczenie 6.2.

Napisz funkcję, która przyjmuje dwa argumenty liczbowe i zwraca większy z nich. Wykonaj dwa warianty ćwiczenia. W pierwszym użyj operatora warunkowego, w drugim — nie używaj go.

Ćwiczenie 6.3.

Napisz funkcję przyjmującą trzy argumenty stanowiące parametry równania kwadratowego. Wyświetl w oknie dialogowym wszystkie rozwiązania tego równania, informację o braku rozwiązania lub też, jeśli parametry nie określają równania kwadratowego, komunikat o tym.

Ćwiczenie 6.4.

Napisz funkcję potega podnoszącą pierwszy jej argument do potęgi całkowitej dodatniej określonej przez drugi argument. Funkcja powinna zwracać wynik potęgowania. Przykładowo wywołanie potega(2, 8) powinno dać w wyniku wartość 256 (nie używaj obiektu Math).

Ćwiczenie 6.5.

Napisz funkcję przyjmującą dwa argumenty i zwracającą ich różnicę, gdy pierwszy jest nieujemny, lub ich sumę, gdy pierwszy jest ujemny. Obliczenie sumy i różnicę powinno zostać wykonane przez pomocnicze funkcje wewnętrzne.

Rozdział 3.

Obiekty, tablice i wyjątki

Lekcja 7. Standardowe obiekty i funkcje

Opisane do tej pory konstrukcje pozwalają na wykonywanie naprawdę wielu zadań programistycznych. Jednak tworzenie własnych funkcji w celu wykonania takich czynności jak przetwarzanie wyrażeń, kodowanie adresów URL, uzyskiwanie wartości pierwiastków, funkcji trygonometrycznych czy potęg zajęłoby na pewno wiele cennego czasu. Na szczęście, JavaScript oferuje zestaw gotowych procedur, które wykonają zarówno wymienione, jak wiele innych przydatnych zadań. Warto je poznać, czym zajmiemy się w tej właśnie lekcji.

Funkcje globalne

W JavaScriptie możemy korzystać z zestawu funkcji globalnych, czyli takich, które są dostępne w każdym miejscu skryptu. Zostały one zebrane w tabeli 3.1 i pozwalają na wykonywanie różnych przydatnych operacji. Niektórym z nich przyjrzymy się nieco bliżej.

Przetwarzanie wartości liczbowych

Jedną z często wykonywanych czynności jest konwersja ciągu znaków na wartość liczbową. Jest to niezbędne np. wtedy, gdy użytkownik wprowadza dane do formularza umieszczonego na stronie WWW (sposoby obsługi formularzy zostaną podane w dalszej części książki). Takie dane, nawet jeśli tworzą wartość liczbową, są traktowane jak ciąg znaków. Aby więc użyć tych danych jako liczby, trzeba je najpierw w jakiś sposób przetworzyć (przekonwertować). Pomagają w tym dwie funkcje. Pierwsza z nich to `parseInt`. Pozwala ona na zamianę ciągu znaków na wartość całkowitą. Jej wywołanie ma postać:

```
parseInt(str[, podstawa])
```

Tabela 3.1. Funkcje globalne

Nazwa	Sposób wywołania	Opis	Dostępność
decodeURI	decodeURI(<i>URI</i>)	Dekoduje ciąg znaków <i>URI</i> zakodowany przez funkcję encodeURI.	od JavaScript 1.5 i JScript 5.5
decodeURI- ↳Component	decodeURI- ↳Component(<i>URI</i>)	Dekoduje ciąg znaków <i>URI</i> zakodowany przez funkcję encodeURIComponent.	od JavaScript 1.5 i JScript 5.5
encodeURI	encodeURI(<i>str</i>)	Przekształca ciąg <i>str</i> w taki sposób, aby był poprawnym identyfikatorem zasobów (URI, ang. <i>Uniform Resource Identifier</i>). Kodowanie odbywa się poprzez zamianę wybranych znaków na odpowiadające im sekwencje zgodne z kodowaniem UTF-8. Nie są kodowane znaki :, /, ;, ?.	od JavaScript 1.5 i JScript 5.5
encodeURI- ↳Component	encodeURI- ↳Component(<i>str</i>)	Przekształca ciąg <i>str</i> w taki sposób, aby był poprawnym identyfikatorem zasobów. Kodowanie odbywa się poprzez zamianę wybranych znaków na odpowiadające im sekwencje zgodne z kodowaniem UTF-8. W przeciwieństwie do metody encodeURI, kodowane są również znaki :, /, ;, ?.	od JavaScript 1.5 i JScript 5.5
escape	escape(<i>str</i>)	Zwraca przetworzony ciąg <i>str</i> , w którym wszystkie znaki inne niż znaki alfabetu łacińskiego oraz znaki @, *, -, _, +, ., / zostały zamienione na sekwencje znaków specjalnych. Dla znaków o kodach poniżej 256 sekwencja specjalna ma postać %xx, gdzie xx to heksadecymalny kod znaku, natomiast dla znaków o kodach powyżej 255 sekwencja ma postać %unnnn, gdzie nnnn to kod znaku w formacie Unicode.	od JavaScript 1.0 i JScript 1.0
eval	eval(<i>wyrażenie</i>)	Jeżeli parametr <i>wyrażenie</i> jest ciągiem znaków tworzącym wyrażenie, zwraca jego obliczoną wartość, jeżeli natomiast <i>wyrażenie</i> jest instrukcją lub ciągiem instrukcji, są one wykonywane.	od JavaScript 1.0 i JScript 1.0
isFinite	isFinite ↳(<i>wartość</i>)	Zwraca wartość true, jeżeli parametr <i>wartość</i> przedstawia wartość skończoną, lub false w przeciwnym przypadku.	od JavaScript 1.3 i JScript 3.0
isNaN	isNaN(<i>wartość</i>)	Zwraca wartość false, jeżeli parametr <i>wartość</i> przedstawia wartość liczbową, lub true w przeciwnym przypadku.	od JavaScript 1.0 i JScript 1.0
parseFloat	parseFloat(<i>str</i>)	Przetwarza ciąg znaków podany jako argument <i>str</i> na wartość rzeczywistą. Jeżeli ciąg <i>str</i> nie będzie reprezentował prawidłowej wartości rzeczywistej, funkcja zwróci wartość NaN.	od JavaScript 1.0 i JScript 1.0
parseInt	parseInt(<i>str</i> ↳ [, <i>podstawa</i>])	Przetwarza ciąg znaków podany jako argument <i>str</i> na wartość całkowitą. Opcjonalny argument <i>podstawa</i> pozwala na ustalenie podstawy systemu liczbowego.	od JavaScript 1.0 i JScript 1.0
unescape	unescape(<i>str</i>)	Odwrotność funkcji escape. Zwraca ciąg, w którym wszystkie sekwencje specjalne zostały zamienione na odpowiadające im znaki.	od JavaScript 1.0 i JScript 1.0

Opcjonalny argument *podstawa* pozwala na ustalenie podstawy systemu liczbowego, może przyjmować wartości od 2 do 36. W przypadku kiedy argument *podstawa* zostanie pominięty lub będzie miał wartość 0, przyjęte zostaną następujące zasady:

- ◆ jeżeli ciąg *str* rozpoczyna się od znaków 0x, będzie traktowany jak wartość szesnastkowa,
- ◆ jeżeli ciąg *str* rozpoczyna się od znaku 0, będzie traktowany jak wartość ósemkowa¹,
- ◆ jeżeli nie zachodzą przypadki 1. i 2., ciąg *str* będzie traktowany jak wartość dziesiętna.

Jeżeli ciąg *str* nie będzie reprezentował prawidłowej wartości całkowitej, funkcjawróci wartość specjalną NaN, oznaczającą: „To nie jest liczba” (z ang. *Not a Number*).

Przypomnijmy sobie materiał z lekcji 3. Został w niej poruszony temat dodawania ciągów znaków. Wiemy zatem, że operacja typu:

"123" + "456"

da w wyniku ciąg:

"123456"

Co zrobić, aby te dane zostały potraktowane jak liczby? Trzeba użyć funkcji parseInt. Zobrazowano to na przykładzie widocznym na listingu 3.1.

Listing 3.1. Konwersja ciągu na liczbę całkowitą

```
var str = "";
var ciąg1 = "123";
var ciąg2 = "456";

var wynik = ciąg1 + ciąg2;

str += "Przed konwersją wynikiem dodawania ciąg1 + ciąg2 jest ";
str += wynik;

ciag1 = parseInt(ciąg1);
ciag2 = parseInt(ciąg2);
wynik = ciąg1 + ciąg2;

str += ".<br />";
str += "Po konwersji wynikiem dodawania ciąg1 + ciąg2 jest ";
str += wynik;
str += ".<br />";

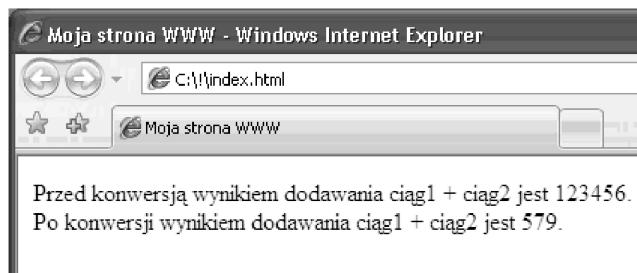
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

¹ Jest to zachowanie charakterystyczne dla interpreterów JavaScirpt dostarczanych w popularnych przeglądarkach. Standard ECMAScript zezwala jednak na traktowanie takiej wartości zarówno jak ósemkowej, jak i dziesiętnej. Jeśli nie mamy pewności, na jakich platformach będzie uruchamiany skrypt, zaleca się jawne stosowanie argumentu określającego system liczbowy.

Na początku skryptu zostały zadeklarowane zmienne ciąg1 i ciąg2, którym zostały przypisane ciągi znaków 123 i 456. W związku z tym, pierwsza operacja dodawania ciąg1 + ciąg2 to w rzeczywistości łączeniełańcuchów znakowych. Wynikiem jest więc ciąg 123456. W dalszej części skryptu za pomocą funkcji parseInt oba łańcuchy zostały zamienione na wartości liczbowe 123 i 456. Stąd druga operacja ciąg1 + ciąg2 jest operacją dodawania arytmetycznego, której wynikiem jest 579. Ostatecznie po uruchomieniu skryptu w przeglądarce ujrzymy widok przedstawiony na rysunku 3.1.

Rysunek 3.1.

Wynik dodawania jest zależny od typu zmiennej



Gdy ciąg znaków, który chcemy przetworzyć na liczbę, zawiera wartość zapisaną w systemie innym niż dziesiętny, używamy drugiego argumentu funkcji parseInt. Poniższa lista zawiera kilka przykładowych wywołań wraz z ich wynikami:

Wywołanie	Wynik
parseInt("10", 10);	10
parseInt("10", 16);	16
parseInt("10", 2);	2
parseInt("10", 16);	28

W zapisie szesnastkowym można stosować zarówno małe, jak i wielkie litery (od *a* do *f*). Należy też pamiętać, że funkcja parseInt przetwarza ciąg znaków aż do napotkania pierwszego znaku spoza danego systemu liczbowego. Wtedy pierwsze znaki są przetwarzane na liczbę, a pozostała część ciągu — odrzucana. Przykładowo (odrzucone fragmenty ciągów zostały wyróżnione):

Wywołanie	Wynik
parseInt("123abc", 10);	123
parseInt("1ARD", 16);	26
parseInt("115", 2);	3
parseInt("243", 2);	NaN

W podobny sposób działa funkcja parseFloat, z tym że przetwarza ciąg znaków podany jako argument na wartość rzeczywistą. Jeżeli ciąg ten nie będzie reprezentował prawidłowej wartości rzeczywistej, funkcja zwróci wartość NaN. Przykład zastosowania parseFloat został zaprezentowany na listingu 3.2.

Listing 3.2. Użycie funkcji *parseFloat*

```
var str = "";
var ciąg1 = "12.3";
var ciąg2 = "4.56E2";

var wynik = ciąg1 + ciąg2;

str += "Przed konwersją wynikiem dodawania ciąg1 + ciąg2 jest ";
str += wynik;

ciag1 = parseFloat(ciąg1);
ciag2 = parseFloat(ciąg2);
wynik = ciąg1 + ciąg2;

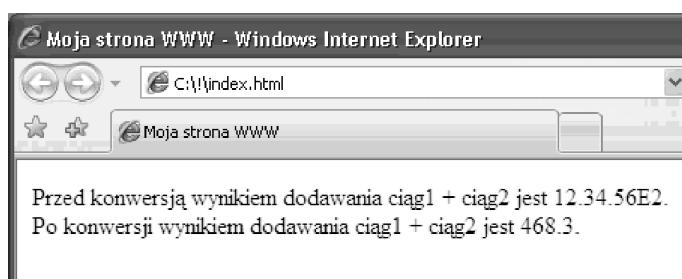
str += ".<br />";
str += "Po konwersji wynikiem dodawania ciąg1 + ciąg2 jest ";
str += wynik;
str += ".<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Struktura skryptu jest taka sama jak w kodzie z listingu 3.1. Tworzone są dwie zmienne: ciąg1 i ciąg2, zawierające ciągi znaków 12.3 oraz 4.56E2. Pierwsza operacja dodawania ciąg1 + ciąg2 będzie w istocie konkatenacją ciągów znakowych, a więc w wyniku otrzymamy ciąg 12.34.56E2. Dopiero po użyciu funkcji *parseFloat* oba ciągi zostaną przetworzone na wartości liczbowe. W pierwszym przypadku będzie to oczywiście 12,3. Przypadek drugi może być mniej oczywisty, gdyż będziemy mieli do czynienia z wartością 456. Zapis 4.56E2 to bowiem notacja wykładnicza oznaczająca $4,56 \cdot 10^2$. A zatem w wyniku drugiego dodawania otrzymamy wartość 468,3. Dlatego też po uruchomieniu skryptu ujrzymy widok zaprezentowany na rysunku 3.2.

Rysunek 3.2.

Konwersja typów zmienia wynik dodawania



Podczas wykonywania konwersji i innych operacji na liczbach warto zwrócić uwagę na dwie przydatne funkcje: *isNaN* oraz *isFinite*. Pierwsza zwraca wartość true, jeżeli argument ma wartość specjalną NaN lub w wyniku konwersji na liczbę da on wartość NaN; w przeciwnym razie zwraca false. Jest wyjątkowo przydatna. Co się bowiem stanie, jeśli np. w wyniku działania *parseInt* nie powstanie wartość liczbową? Zgodnie z tym, co zostało napisane powyżej, funkcja ta zwróci wartość NaN. W jakiś sposób trzeba jednak taką sytuację rozpoznać. Wtedy właśnie należy użyć *isNaN*, np.:

```

var x = parseInt("abc");
if(isNaN(x)){
    alert("x nie jest liczbą!");
}
else{
    alert("Konwersja prawidłowa. x jest liczbą.");
}

```

Ogólnie można powiedzieć, że wartość NaN powstaje w wyniku wszelkich nieprawidłowych operacji arytmetycznych (np. dzielenia przez 0).

Funkcja isFinite zwraca true, jeśli argument ma wartość skońzoną lub może zostać przekonwertowany na wartość skońzoną. Wartość false zwraca natomiast w przeciwnym przypadku, czyli wtedy, gdy testowany argument ma wartość NaN lub reprezentuje dodatnią bądź ujemną nieskończoność.

Kodowanie ciągów znaków

Przy programowaniu stron WWW często zachodzi konieczność przekazania danych w adresie URL. Jednak taki adres ma ściśle określoną postać. Mogą się w nim znaleźć tylko znaki alfabetu łacińskiego (litery ASCII), cyfry oraz część znaków przestankowych i interpunkcyjnych. Takie ograniczenie nie może być zaakceptowane, trzeba więc znaleźć jakiś sposób na przekazanie również innych znaków (bajtów). W tym celu należy użyć funkcji z tabeli 3.1. Jeśli chcemy utworzyć pełny adres URL, powinniśmy użyć funkcji encodeURI. Działa ona w taki sposób, że wszystkie znaki niestandardowe (czyli te, które nie mogą się znaleźć adresie) zamienia na odpowiadające im kody Unicode, a następnie zapisuje je w formacie szesnastkowym. Funkcja nie zmienia znaków: -, _, ., !, ~, *, ', (,), :, /, ?, :, @, &, =, +, \$, ., #. Jak to wyglądałoby w praktyce? Założmy, że przygotowaliśmy w skrypcie adres:

<http://mojadomena.com/dane.php?tekst=dane testowe>

Zauważmy, że parametr tekst zawiera串 znaków, w którym jest spacja. Taki adres jest nieprawidłowy. Jeśli jednak użyjemy funkcji encodeURI:

`encodeURI("http://mojadomena.com/dane.php?tekst=dane testowe");`

zwróci zostanie prawidłowo zakodowany adres w postaci:

<http://mojadomena.com/dane.php?tekst=dane%20testowe>

Jak widać, spacja została zamieniona na串 %20. Jest to kod tego znaku poprzedzony znakiem %. Podobnie, jeśli w danych znajdą się polskie znaki, również niezbędne będzie ich zakodowanie. Przykładowo串:

<http://mojadomena.com/dane.php?tekst=piękna Łąka>

zostanie zamieniony na:

<http://mojadomena.com/dane.php?tekst=pi%C4%99ka%20%C5%82%C4%85ka>

Tak przetworzone adresy URL będą już bez problemów interpretowane przez przeglądarkę i serwer WWW.

Druga funkcja to encodeURIComponent. Jest przeznaczona do kodowania jedynie fragmentów adresu, np. samych danych (wartości parametrów). W tym bowiem przypadku kodowaniu podlegają również znaki: :, /, ?, :, @, &, =, +, \$, ., #. Funkcję tę można by więc wykorzystać następująco:

```
var url = "http://mojadomena.com/dane.php?";  
var tekst = "piękna Łąka":  
  
tekst = encodeURIComponent(tekst);  
url += tekst;
```

Ciąg piękna Łąka zostanie zamieniony na pi%C4%99kna%20%C5%82%C4%85ka, a następnie dodany do adresu URL znajdującego się w zmiennej url. Ostatecznie powstanie taki sam adres jak w poprzednim przypadku.

Oczywiście, musimy być przygotowani na sytuacje odwrotne, tzn. możemy otrzymać adres URL lub same dane zakodowane w przedstawiony sposób i trzeba będzie je rozkodować. Tu pomagają funkcje decodeURI i decodeURIComponent. Pierwsza powinna być używana, jeśli chcemy zdekodować pełny adres URL, a druga — gdy chcemy zdekodować inne dane (czyli takie, które nie stanowią adresu). Jeśli więc mamy adres w postaci:

<http://mojadomena.com/dane.php?tekst=pi%C4%99kna%20%C5%82%C4%85ka>

to po użyciu funkcji decodeURI zostanie on zamieniony na:

<http://mojadomena.com/dane.php?tekst=piękna Łąka>

W tym miejscu musimy też wspomnieć o funkcjach escape i unescape. Otóż, są to pierwotyczne funkcje opisanych wyżej i były stosowane w wersjach ECMAScript poniżej wersji 3. Jednak począwszy od ECMAScript 3 (JavaScript 1.5, JScript 5.5), zostały uznane za przestarzałe i należy unikać ich stosowania. Ponieważ jednak w wielu skryptach wciąż są spotykane, omówimy je skrótnie.

Funkcja escape zamienia wszystkie znaki inne niż litery ASCII, cyfry i znaki @, *, _, +, -, ., / na sekwencje specjalne (inaczej sekwencje ucieczki — stąd też nazwa funkcji) %nn lub %nnnn, gdzie nn oznacza kod znaku. Sekwencja %nn jest używana dla znaków Unicode o kodach od \u0000 do \u00ff, a sekwencja %nnnn dla wszystkich pozostałych. Przykładowo znany już tekst „piękna Łąka” po przetworzeniu przez funkcję escape będzie miał postać:

pi%u0119kna%20%u0142%u0105ka

Funkcja unescape ma działanie odwrotne, tzn. zamienia sekwencje specjalne na odpowiadające im znaki. A zatem użycie jej w stosunku do powyższego tekstu spowoduje przetworzenie go z powrotem na zrozumiałą formę: piękna Łąka.

Przetwarzanie wyrażeń

Z tabeli 3.1. znajdziemy bardzo ciekawą funkcję eval. Służy ona do przetwarzania wyrażeń i jest dostępna już od najwcześniejszych wersji języka JavaScript. Co oznacza przetwarzanie wyrażeń? Otóż, jeśli funkcji w postaci argumentu przekażemy ciąg

znaków tworzących wrażenie arytmetyczne, zwróci jego wynik. Jeżeli natomiast argument będzie instrukcją lub ciągiem instrukcji, zostaną one wykonywane. Jeśli w tym przypadku ostatnia instrukcja zwróciła jakąś wartość, wartość ta zostanie również zwrócona przez eval; jeżeli natomiast ostatnia instrukcja nie zwróciła żadnej wartości, wartością zwróconą przez eval będzie undefined. Jeśli np. jako argumentu użyjemy ciągu:

```
(6 + 7 * 2) / 5
```

wynikiem działania funkcji będzie wartość 4; jeśli argumentem będzie ciąg func(), a w skrypcie będzie dostępna funkcja o nazwie func, zostanie ona wykonana. Spójrzmy na listing 3.3.

Listing 3.3. Przykłady użycia funkcji eval

```
function func()
{
    alert("To jest funkcja func.");
    return 0;
}

var str = "";

var ciąg1 = "(6 + 7 * 2) / 5";
var ciąg2 = "alert(\"abc\")";

var wynik1 = eval(ciąg1);
var wynik2 = eval(ciąg2);
var wynik3 = eval("func():");

str += "Wynikiem <b>eval(" + ciąg1 + ")</b> jest " + wynik1;
str += ".<br />";
str += "Wynikiem <b>eval(" + ciąg2 + ")</b> jest " + wynik2;
str += ".<br />";
str += "Wynikiem <b>eval(\"func():\")</b> jest " + wynik3;
str += ".<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Na początku została zdefiniowana funkcja o nazwie func. Ma ona bardzo prostą konstrukcję. Zawiera instrukcję alert wyświetlającą okno dialogowe ze zdefiniowanym komunikatem oraz instrukcję return, która powoduje zwrócenie wartości 0 (funkcja zwraca więc 0 jako wynik swojego działania). Za funkcją znajdują się deklaracje dwóch zmiennych: ciąg1 i ciąg2. Pierwszej został przypisany ciąg znaków zawierający wymienione wyżej wyrażenie arytmetyczne, drugiej — ciąg znaków stanowiący instrukcję JavaScript (wywołanie funkcji alert). Zwróćmy uwagę na sposób zapisu wywołania funkcji alert:

```
var ciąg2 = "alert(\"abc\")";
```

Ponieważ jako argument otrzymuje ona ciąg znaków abc, który musi być ujęty w znaki cudzysłowu (lub apostrofu), niezbędnego było użycie sekwencji specjalnej \" (lekcja 2.). Alternatywnie można użyć znaków apostrofu, dzięki czemu zapis byłby bardziej czytelny:

```
var ciąg2 = "alert('abc')";
```

Za definicjami zmiennych wywoływana jest trzykrotnie funkcja eval, a wyniki tych wywołań są przypisywane zmiennym wynik1, wynik2 i wynik3. Pierwsze wywołanie ma postać:

```
eval(ciąg1);
```

Ponieważ zmienna ciąg1 zawiera wyrażenie arytmetyczne, zostanie ono obliczone, a wynik (wartość 4) zostanie zwrócony jako rezultat wywołania funkcji.

Drugie wywołanie jest bardzo podobne, z tą różnicą, że jako argument została zastosowana zmienna ciąg2:

```
eval(ciąg2);
```

Ponieważ ta zmienna zawiera instrukcję wywołującą funkcję alert, zostanie ona wykonana, a na ekranie pojawi się okno dialogowe z napisem abc. Ze względu na to, że funkcja alert nie zwraca żadnej wartości, wynikiem działania eval będzie tym razem wartość undefined.

Trzecie wywołanie jest nieco inne:

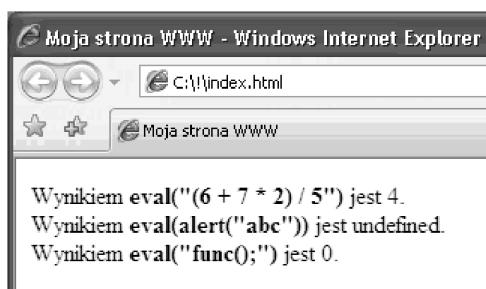
```
eval("func()");
```

Jako argument został tu przekazany bezpośrednio ciąg znaków stanowiący instrukcję wywołującą funkcję func. Zostaną więc wykonane znajdujące się w jej wnętrzu instrukcje, czyli alert i return. Na ekranie pojawi się zatem kolejne okno dialogowe. Ponieważ została użyta instrukcja return i funkcja func zwróciła wynik (wartość 0), będzie on też rezultatem działania funkcji eval.

Po wykonaniu wymienionych instrukcji konstruowany jest opis podający wartości zwrócone przez każde z wymienionych wywołań. A zatem w rezultacie działania skryptu na ekranie pojawią się dwa okna dialogowe, a następnie treść widoczna na rysunku 3.3.

Rysunek 3.3.

Komunikat podający wartości zwrócone przez wywołania funkcji eval



Właściwości globalne

Oprócz globalnych funkcji, istnieją również globalne właściwości. W rzeczywistości są to definicje typów Infinity, NaN i undefined. Zostały one zebrane w tabeli 3.2².

Tabela 3.2. Właściwości globalne

Właściwość	Znaczenie	Dostępność
Infinity	Reprezentuje nieskończoność. Jest tożsama z wartością właściwości POSITIVE_INFINITY obiektu Number.	Od JavaScript 1.3 i JScript 3.0
NaN	Reprezentuje nieliczbę. Jest tożsama z wartością właściwości NaN obiektu Number.	Od JavaScript 1.3 i JScript 3.0
undefined	Reprezentuje wartość niezdefiniowaną.	Od JavaScript 1.3 i JScript 5.5

Nieco matematyki (obiekt Math)

Tworząc skrypty, nie uciekniemy od matematyki, zawsze trzeba będzie coś policzyć, czy zastosować pewien wzór, choćby po to, aby wyliczyć współrzędne, w jakich ma się pojawić dany element witryny. Co prawda, potrafimy już korzystać z podstawowych operacji arytmetycznych, ale one nie zawsze będą wystarczające. Z pomocą przyjdzie nam globalny obiekt Math. Co oznacza termin „obiekt”, dowiemy się dopiero w kolejnej lekcji, na razie potraktujmy go jako zbiór wartości i funkcji. Wartości przechowywane przez obiekt nazwiemy właściwościami, a funkcje — metodami. Spójrzmy na tabelę 3.3, która zawiera zestaw właściwości obiektu Math. Jeśli chcemy użyć w skrypcie liczby pi, piszemy Math.PI, a jeśli stałej Eulera — Math.E.

Napiszemy teraz skrypt wyświetlający na stronie wartości wszystkich właściwości z tabeli 3.3 oraz wykonujący kilka operacji arytmetycznych. Kod takiego skryptu będzie miał postać widoczną na listingu 3.4, natomiast efekt jego wykonania będzie taki, jak na rysunku 3.4.

Pierwsza część skryptu po prostu odczytuje stan kolejnych właściwości zapisanych w obiekcie Math i dopisuje ich wartości do zmiennej str wraz z odpowiednimi opisami. Dowiemy się więc np., że:

```
E = 2.718281828459045
```

W drugiej części właściwości reprezentujące stałe matematyczne są używane w trzech działaniach arytmetycznych. Wartość pi jest mnożona przez 2:

```
var wynik1 = 2 * Math.PI;
```

a logarytm naturalny z 2 jest podnoszony do drugiej potęgi (mnożony przez siebie):

```
var wynik2 = Math.LN2 * Math.LN2;
```

² Pominięte zostały właściwości java i Packages związane ze współpracą z pakietami języka Java, które nie są omawiane w tej publikacji.

Tabela 3.3. Właściwości obiektu Math

Nazwa	Znaczenie	Składnia	Wartość	Dostępność
E	Stała Eulera	Math.E	ok. 2,718	od JavaScript 1.0 i JScript 1.0
LN2	Logarytm naturalny z 2	Math.LN2	ok. 0,693	od JavaScript 1.0 i JScript 1.0
LN10	Logarytm naturalny z 10	Math.LN10	ok. 2,302	od JavaScript 1.0 i JScript 1.0
LOG2E	Logarytm o podstawie 2 z E	Math.LOG2E	ok. 1,442	od JavaScript 1.0 i JScript 1.0
LOG10E	Logarytm o podstawie 10 z E	Math.LOG10E	ok. 0,434	od JavaScript 1.0 i JScript 1.0
PI	Liczba pi (π)	Math.PI	ok. 3,14159	od JavaScript 1.0 i JScript 1.0
SQRT1_2	Pierwiastek kwadratowy z 0,5	Math.SQRT1_2	ok. 0,707	od JavaScript 1.0 i JScript 1.0
SQRT2	Pierwiastek kwadratowy z 2	Math.SQRT2	ok. 1,414	od JavaScript 1.0 i JScript 1.0

Listing 3.4. Użycie stałych matematycznych

```

var str = "";

str += "E = " + Math.E + "<br />";
str += "LN2 = " + Math.LN2 + "<br />";
str += "LN10 = " + Math.LN10 + "<br />";
str += "LOG2E = " + Math.LOG2E + "<br />";
str += "LOG10E = " + Math.LOG10E + "<br />";
str += "PI = " + Math.PI + "<br />";
str += "SQRT1_2 = " + Math.SQRT1_2 + "<br />";
str += "SQRT2 = " + Math.SQRT2 + "<br /><br />";

var wynik1 = 2 * Math.PI;
var wynik2 = Math.LN2 * Math.LN2;
var wynik3 = Math.SQRT2 * Math.SQRT2;

str += "2 * pi = " + wynik1 + "<br />";
str += "ln<sup>2</sup>(2) = " + wynik2 + "<br />";
str += "(&radic;2)<sup>2</sup> = " + wynik3 + "<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

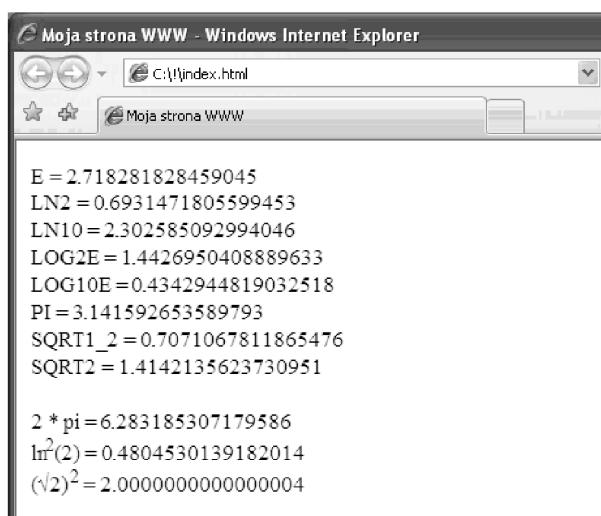
To samo działanie jest też wykonywane w stosunku do wartości pierwiastka kwadratowego (drugiego stopnia) z 2:

```
var wynik3 = Math.SQRT2 * Math.SQRT2;
```

Wyniki tych działań są również wyświetlane na ekranie, co widać na rysunku 3.4. Indeks górny w opisach działań uzyskujemy dzięki znacznikowi $\langle\sup\rangle$, natomiast symbol pierwiastka przy użyciu encji $\√$. Zwróćmy jednak uwagę na wynik trzeciego działania. Moglibyśmy się spodziewać, że jeśli podniesiemy do drugiej potęgi wartość pierwiastka kwadratowego z 2, otrzymamy wartość 2. Tymczasem wynikiem jest 2.0000000000000004 . Dlaczego? Wszystkiemu winna jest niedokładność reprezentacji. Nie mamy bowiem do czynienia z sytuacją idealną, a wartość pierwiastka jest niewymierna i reprezentowana z pewną dokładnością (liczba miejsc po przecinku jest ściśle określona). Jeśli na takiej niedokładnej wartości wykonamy operację arytmetyczną (szczególnie jeśli drugim argumentem jest również wartość niewymierna, tak jak w omawianym przypadku), powiększymy błąd związany z niedokładnością reprezentacji. Dlatego też na 16. miejscu po przecinku pojawiła się wartość 4, a nie — jak byśmy tego oczekiwali — 0. Warto na tę kwestię zwrócić uwagę, kiedy wykonujemy operacje na liczbach rzeczywistych, a szczególnie wtedy, gdy porównujemy takie wartości. Wyniki mogą się bowiem różnić od naszych oczekowań.

Rysunek 3.4.

Wartości stałych matematycznych oraz wyniki wykonanych na nich działań



The screenshot shows a Windows Internet Explorer window titled "Moja strona WWW - Windows Internet Explorer". The address bar displays "C:\index.html". The page content lists various mathematical constants and their decimal values:

```

E = 2.718281828459045
LN2 = 0.6931471805599453
LN10 = 2.302585092994046
LOG2E = 1.4426950408889633
LOG10E = 0.4342944819032518
PI = 3.141592653589793
SQRT1_2 = 0.7071067811865476
SQRT2 = 1.4142135623730951

2 * pi = 6.283185307179586
ln2(2) = 0.4804530139182014
(√2)2 = 2.0000000000000004

```

Same stałe matematyczne to trochę za mało. Obiekt `Math` udostępnia więc również zestaw funkcji matematycznych. Z łatwością policzymy wartość sinusa, czy podniesiemy dowolną wartość do dowolnie wybranej potęgi³. Lista funkcji matematycznych została zebrana w tabeli 3.4.

Jednej z tych funkcji matematycznych już wcześniej użyliśmy. To `Math.sqrt` obliczająca pierwiastek kwadratowy z podanego argumentu. Jednak również wiele pozostałych przyda się w praktyce programistycznej. Pamiętajmy jednak, że — podobnie jak było to w przypadku stałych — operacje arytmetyczne wykonywane na wartościach rzeczywistych mogą dawać wyniki przybliżone, co wynika z niedokładności reprezentacji tych liczb i konieczności ich zaokrąglania. Latwo możemy też przekroczyć

³ Pod warunkiem, że wynik takiej operacji zmieści się z zakresie wartości, które mogą być standardowo reprezentowane w JavaScriptie.

Tabela 3.4. Metody obiektu Math

Metoda	Składnia	Opis	Dostępność
abs	Math.abs(<i>wartość</i>)	Zwraca wartość bezwzględną parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
acos	Math.acos(<i>wartość</i>)	Zwraca wartość arcus cosinus parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
asin	Math.asin(<i>wartość</i>)	Zwraca wartość arcus sinus parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
atan	Math.atan(<i>wartość</i>)	Zwraca wartość arcus tangens parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
atan2	Math.atan2(<i>x</i> , <i>y</i>)	Zwraca wartość kąta pomiędzy osią <i>OX</i> a punktem wyznaczanym przez parametry <i>x</i> i <i>y</i> .	od JavaScript 1.0 i JScript 1.0
ceil	Math.ceil(<i>wartość</i>)	Zwraca najmniejszą liczbę całkowitą większą lub równą parametrowi <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
cos	Math.cos(<i>wartość</i>)	Zwraca wartość cosinusa parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
exp	Math.exp(<i>wartość</i>)	Zwraca wartość wynikającą z podniesienia stałej Eulera do potęgi wskazywanej przez parametr <i>wartość</i> (odpowiada to działaniu $e^{\text{wartość}}$).	od JavaScript 1.0 i JScript 1.0
floor	Math.floor(<i>wartość</i>)	Zwraca największą liczbę całkowitą mniejszą lub równą parametrowi <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
log	Math.log(<i>wartość</i>)	Zwraca logarytm naturalny (o podstawie równej stałej Eulera) parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
max	Math.max(<i>x</i> , <i>y</i>)	Zwraca większą z wartości przekazanych w postaci argumentów.	od JavaScript 1.0 i JScript 1.0
min	Math.min(<i>x</i> , <i>y</i>)	Zwraca mniejszą z wartości przekazanych w postaci argumentów.	od JavaScript 1.0 i JScript 1.0
pow	Math.pow(<i>x</i> , <i>y</i>)	Zwraca wartość <i>x</i> podniesioną do potęgi <i>y</i> (jest to odpowiednik działania x^y).	od JavaScript 1.0 i JScript 1.0
random	Math.random()	Zwraca wartość pseudolosową z zakresu 0 – 1.	od JavaScript 1.0 i JScript 1.0
round	Math.round(<i>wartość</i>)	Zwraca wartość wynikającą z zaokrąglenia parametru <i>wartość</i> do najbliższej liczby całkowitej.	od JavaScript 1.0 i JScript 1.0
sin	Math.sin(<i>wartość</i>)	Zwraca wartość sinusa parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
sqrt	Math.sqrt(<i>wartość</i>)	Zwraca pierwiastek kwadratowy parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0
tan	Math.tan(<i>wartość</i>)	Zwraca (w radianach) wartość tangensa parametru <i>wartość</i> .	od JavaScript 1.0 i JScript 1.0

dopuszczalny zakres reprezentowanych wartości, szczególnie przy takich funkcjach jak pow. Wykonajmy zatem kilka przykładowych operacji i wyświetlmy ich wyniki na ekranie. Umożliwi to skrypt widoczny na listingu 3.5.

Listing 3.5. Operacje na liczbach rzeczywistych

```

var str = "";

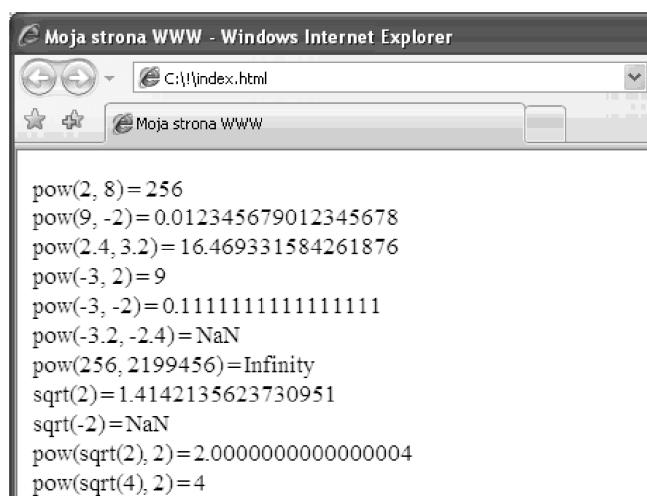
str += "pow(2, 8) = " + Math.pow(2, 8);
str += "<br />pow(9, -2) = " + Math.pow(9, -2);
str += "<br />pow(2.4, 3.2) = " + Math.pow(2.4, 3.2);
str += "<br />pow(-3, 2) = " + Math.pow(-3, 2);
str += "<br />pow(-3, -2) = " + Math.pow(-3, -2);
str += "<br />pow(-3.2, -2.4) = " + Math.pow(-3.2, -2.4);
str += "<br />pow(256, 2199456) = " + Math.pow(256, -2199456);
str += "<br />sqrt(2) = " + Math.sqrt(2);
str += "<br />sqrt(-2) = " + Math.sqrt(-2);
str += "<br />pow(sqrt(2), 2) = " + Math.pow(Math.sqrt(2), 2);
str += "<br />pow(sqrt(4), 2) = " + Math.pow(Math.sqrt(4), 2);

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Efekt działania kodu jest widoczny na rysunku 3.5. Pierwsze instrukcje wykonują różne rodzaje potęgowania. Jak widać, funkcja pow jest bardzo elastyczna i może przyjmować najróżniejsze argumenty. W pierwszym przypadku są to dwie całkowite liczby dodatnie (operacja 2^8), a w drugim — dodatnia i ujemna (operacja 9^{-2}). W przypadku trzecim mamy dwie liczby rzeczywiste, jest to więc podniesienie wartości 2,4 do potęgi 3,2 (operacja $2,4^{3,2}$). Przypadki czwarty i piąty to użycie wartości ujemnych (operacje -3^2 i -3^{-2}). Wszystkie te działania dadzą prawidłowy wynik. Dopiero działanie $\text{Math.pow}(-3.2, -2.4)$ nie zostanie wykonane, a zwróconą wartością będzie NaN. Nie wynika to jednak z niedoskonałości funkcji pow, ale z tego, że wyniku takiego działania nie ma w zbiorze liczb rzeczywistych. Problem występuje też z działaniem $\text{Math.pow}(256, 2199456)$. Tutaj użyliśmy zdecydowanie zbyt wielkiego wykładnika potęgi (operacja $256^{2199456}$), a zatem wynik przekracza dopuszczalny zakres wartości rzeczywistych. Dlatego też rezultatem działania jest wartość specjalna Infinity, oznaczająca nieskończoność.

Rysunek 3.5.
Wyniki użycia funkcji matematycznych



Pierwiastek kwadratowy z dwóch (`Math.sqrt(2)`) został policzony bez problemów, natomiast wynikiem operacji `Math.sqrt(-2)` jest wartość NaN. Powodem jest to, że wyniku tego działania nie ma w zbiorze liczb rzeczywistych. To nie powinno budzić naszego zdziwienia. Zwróćmy jednak uwagę na dwie ostatnie operacje. Pierwsza z nich to:

```
Math.pow(Math.sqrt(2), 2)
```

czyli obliczenie za pomocą funkcji `sqrt` pierwiastka kwadratowego z 2, a następnie podniesienie go za pomocą funkcji `pow` do drugiej potęgi. Z matematycznego punktu widzenia, wykonujemy więc taką samą operację, jaką w poprzednim przykładzie była `Math.SQRT2 * Math.SQRT2`. Wynik też jest identyczny, choć niezgodny z zasadami matematyki. Co jednak ciekawe, kolejna operacja:

```
Math.pow(Math.sqrt(4), 2);
```

daje już prawidłowy rezultat. Jest to użycie funkcji `sqrt` do uzyskania pierwiastka kwadratowego z 4, a następnie funkcji `pow` do podniesienia go do drugiej potęgi. Czemu wynik jest właściwy? Dlatego, że rezultat działania `sqrt(4)` jest wartością całkowitą, a dokładniej wymierną, o precyzji nieprzekraczającej precyzji wartości rzeczywistych w JavaScriptie. Dlatego też w tym przypadku nie ma problemów z precyzją i zaokrąglaniem.

Trzy funkcje z tabeli 3.4 wykonują zaokrąglanie wartości rzeczywistych do całkowitych. Zaokrąglaniem klasycznym, czyli do najbliższej liczby całkowitej, zajmuje się funkcja `round`, np.:

- ◆ `Math.round(2.49)` da w wyniku wartość 2;
- ◆ `Math.round(2.51)` da w wyniku wartość 3;
- ◆ `Math.round(-2.51)` da w wyniku wartość -3;
- ◆ `Math.round(-2.49)` da w wyniku wartość -2.

Funkcja `ceil` wykonuje operację, którą nazywamy zaokrągleniem w góre, czyli do najbliższej wyższej liczby całkowitej, np.:

- ◆ `Math.ceil(2.49)` da w wyniku wartość 3;
- ◆ `Math.ceil(2.51)` da w wyniku wartość 3;
- ◆ `Math.ceil(-2.51)` da w wyniku wartość -2;
- ◆ `Math.ceil(-2.49)` da w wyniku wartość -2.

Funkcja `floor` wykonuje operację, którą nazywamy zaokrągleniem w dół, czyli do najbliższej niższej liczby całkowitej, np.:

- ◆ `Math.floor(2.49)` da w wyniku wartość 2;
- ◆ `Math.floor(2.51)` da w wyniku wartość 2;
- ◆ `Math.floor(-2.51)` da w wyniku wartość -3;
- ◆ `Math.floor(-2.49)` da w wyniku wartość -3.

Kolejną funkcją, która przyda się w praktyce programistycznej, jest random, często bowiem potrzebujemy uzyskać wartość losową. Należy tylko pamiętać, że funkcja ta zwraca wartość losową z zakresu $<0.0, 1.0)$, czyli wartość rzeczywistą z zakresu zaczynającego się od 0, a kończącego na 1, ale nie włącznie z 1. Z reguły, poszukujemy jednak wartości całkowitej z przedziału, który chcielibyśmy sami określić. Co zrobić? Zastosować funkcję floor i wzór:

```
Math.floor(Math.random() * (n - m + 1)) + m
```

gdzie m to początek przedziału, a n — jego koniec. Aby np. uzyskać wartość losową z przedziału 2 – 10, użyjemy formuły:

```
Math.floor(Math.random() * 9) + 2
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 7.1.

Napisz funkcję przyjmującą jeden argument. Jeśli będzie on mógł być interpretowany jako wartość liczbową, funkcja ma zwrócić jego podwojoną wartość, jeśli zaś nie będzie liczbą — ciąg wynikający z połączenia tego argumentu z nim samym. Przykładowo dla argumentu równego 23 wartością zwracaną będzie 46, a dla argumentu abc — ciąg abcabc.

Ćwiczenie 7.2.

Napisz funkcję przyjmującą dwa argumenty (powinny one reprezentować liczby całkowite nieujemne) i zwracającą wartość losową z wyznaczonego przez nie przedziału. Pamiętaj o zbadaniu, czy argumenty reprezentują prawidłowy przedział liczbowy.

Ćwiczenie 7.3.

Napisz funkcję przyjmującą dwa argumenty typu rzeczywistego. Jeśli pierwszy ma większą wartość, funkcja powinna zwrócić wartość drugiego zaokrągloną w dół, jeśli drugi ma większą wartość, funkcja powinna zwrócić wartość pierwszego zaokrągloną w górę, a jeśli są równe — wartość dowolnego z nich zaokrągloną zgodnie z regułami klasycznymi (do najbliższej liczby całkowitej).

Lekcja 8. Tworzenie obiektów

Lekcja 8. poświęcona jest obiektom, ich tworzeniu i wykorzystywaniu w JavaScriptie. Dowiemy się zatem, czym jest obiekt, w jaki sposób może powstać i z czego się składać; a także o tym, że może przechowywać dane i wykonywać na nich różne operacje. Poznamy także sposób zapisu określany jako JSON oraz nowy rodzaj pętli, pętle for...in, której opis został pominięty w rozdziale 2.

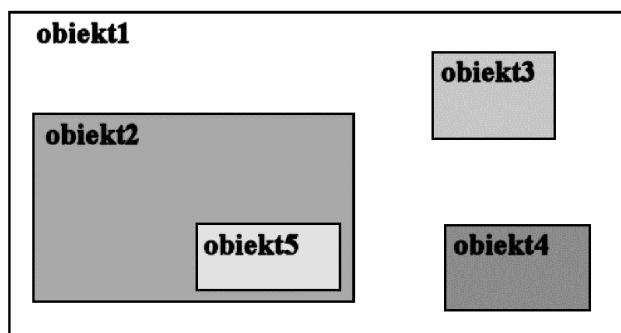
Czym jest obiekt?

Odpowiedź na pytanie, czym są obiekty, nie wydaje się skomplikowana. Chyba każdy intuicyjne rozumie to pojęcie. Obiektem może być praktycznie wszystko: dom, samochód, ale także osoba czy nawet pojedynczy atom. W programowaniu jest podobnie — obiektem możemy nazwać każdy abstrakcyjny byt, który jako programiści zaprogramujemy utworzyć w pamięci komputera. Jeśli np. chcemy przechowywać informacje dotyczącą współrzędnych pewnego punktu na ekranie, możemy utworzyć obiekt, który właśnie takie dane będzie zawierał. Jeszcze bardziej upraszczając to zagadnienie, można powiedzieć, że obiekt to coś, co może przechowywać dane oraz wykonywać różne operacje.

Musimy także wiedzieć, że jeden obiekt może zawierać inne obiekty — dokładnie tak, jak w świecie realnym. Przecież wewnątrz przykładowego samochodu mamy np. fotele, kierownicę, silnik — to są obiekty składowe, podobnie w domu umieszczone są meble, okna, sprzęt AGD. Takie zawieranie się obiektów możemy schematycznie przedstawić tak, jak na rysunku 3.6.

Rysunek 3.6.

Każdy obiekt
może zawierać inne
obiekty składowe



Tworzenie prostych obiektów (JSON)

W najprostszym przypadku możemy potraktować obiekt jako zbiór danych nazywanych właściwościami, a nawet jako zbiór zmiennych. Obiekty takie możemy utworzyć, stosując notację JSON⁴. Skrót ten pochodzi od słów *JavaScript Object Notation*, czyli „notacji obiektów JavaScript” — nazwę JSON należy wymawiać jak angielskie imię Jason (fonetycznie „dżejson”).

Obiekt w notacji JSON trzeba umieścić między znakami nawiasu klamrowego:

```
{  
    tu definicja obiektu  
}
```

⁴ Dokładny opis notacji JSON można znaleźć m.in. pod adresem <http://www.json.org/>.

Może on zawierać różne elementy składowe. Najczęściej są to właściwości definiowane jako pary nazwa-wartość, ogólnie:

```
nazwa_właściwości:wartość_właściwości
```

Nazwa powinna być ciągiem znaków ujętym w cudzysłów, natomiast wartością może być ciąg znaków, liczba, obiekt, tablica (lekция 10.), słowa true i false lub wartość pusta null. Poszczególne właściwości należy oddzielać znakami przecinka. Ogólnie obiekt z właściwościami będzie miał postać:

```
{  
    nazwa_właściwości_1:wartość_właściwości_1,  
    nazwa_właściwości_2:wartość_właściwości_2,  
  
    nazwa_właściwości_N:wartość_właściwości_N  
}
```

Właściwościami obiektu mogą być również inne obiekty definiowane w nawiasie klamrowym:

```
{  
    nazwa_obiektu_składowego:{  
        właściwości obiektu składowego  
    }  
}
```

Mogą to być również tablice definiowane za pomocą nawiasu kwadratowego, ogólnie:

```
{  
    tablica:[  
        wartość1,  
        wartość2,  
        ...  
        wartośćN  
    ]  
}
```

Oto przykład:

```
{  
    imiona:[  
        "Magda",  
        "Maria",  
        "Monika"  
    ]  
}
```

Jak wyglądałby kod obiektu w standardzie JSON przechowującego podstawowe dane hipotetycznej osoby? Przy założeniu, że przechowywałby informacje o imieniu, nazwisku i adresie, przy czym dane adresowe powinny stanowić osobny obiekt, zagnieżdżony w głównym obiekcie, kod przyjąłby postać widoczną na listingu 3.6.

Listing 3.6. Przykładowy obiekt w standardzie JSON

```
{  
    "imię": "Anna",  
    "nazwisko": "Kowalska",  
    "adres": {  
        "ulica": "Malinowa 85",  
        "miasto": "Gliwice"  
    }  
}
```

Definicja zaczyna się od znaku {, a kończy znakiem }. Wewnątrz znajdują się trzy właściwości: imię, nazwisko i adres. Pierwsze dwie mają wartości tekstowe określające imię i nazwisko osoby, odpowiednio Anna i Kowalska, natomiast trzecia właściwość (adres) jest obiektem, który posiada dwie własne właściwości określające ulicę i miasto. Definicja obiektu adres jest umieszczona między znakami nawiasu klamrowego.

Jak skorzystać z takiego obiektu w skrypcie? Mamy dwie możliwości. Pierwszy to deklaracja zmiennej i przypisanie jej definicji obiektu w standardzie JSON, schematycznie:

```
var obj = {  
    //definicja obiektu  
};
```

A to przykład:

```
var osoba = {  
    "imię": "Anna",  
    "nazwisko": "Kowalska",  
}
```

Drugi sposób to użycie funkcji eval. Jest konieczny, gdy obiekt nie jest bezpośrednio zdefiniowany w kodzie, ale został otrzymany jako dane, np. przez transmisję z użyciem AJAX-a⁵. Wtedy dane są traktowane jak tekst i trzeba je przetworzyć na obiekt. W tym celu poddaje się je działaniu omówionej w poprzedniej lekcji funkcji eval. Należy jedynie pamiętać, że przed definicją obiektu trzeba dodatkowo postawić znak (, a za definicją znak). Tak więc przy założeniu, że w zmiennej obj_tekst znajduje się tekstowa definicja, utworzenie obiektu i przypisanie go zmiennej obj osiągniemy, wykonując instrukcję:

```
var obj = eval("(" + obj_tekst + ")");
```

Kiedy obiekt jest gotowy i istnieje w kodzie skryptu, możemy się odwoływać do zawartych w nim danych (właściwości). Robimy to, korzystając z operatora . (znak kropki). Wartość danej właściwości można odczytać za pomocą schematycznej konstrukcji:

```
var zmienna = nazwa_obiektu.nazwa_właściwości;
```

⁵ Czytelnicy zainteresowani tworzeniem stron WWW w technologii AJAX powinni zapoznać się z publikacjami *AJAX. Praktyczny kurs* (<http://helion.pl/ksiazki/ajaxpk.htm>) i *AJAX. Ćwiczenia* (<http://helion.pl/ksiazki/cajax.htm>).

Jeśli zatem chcemy odczytać wartość właściwości imię obiektu osoba i przypisać ją zmiennej imię_osoby, powinniśmy użyć instrukcji:

```
var imię_osoby = osoba.imię;
```

Oczywiście, nie ma konieczności przypisywania wartości właściwości zmiennej, można jej użyć bezpośrednio, np.:

```
alert(osoba.imię);
```

Przykładowy kod tworzący prosty obiekt i wyświetlający pobrane z niego dane został umieszczony na listingu 3.7.

Listing 3.7. Utworzenie obiektu przy zastosowaniu składni JSON

```
var osoba = {
    "imię": "Anna",
    "nazwisko": "Kowalska",
    "adres": {
        "ulica": "Malinowa 85",
        "miasto": "Gliwice"
    }
}

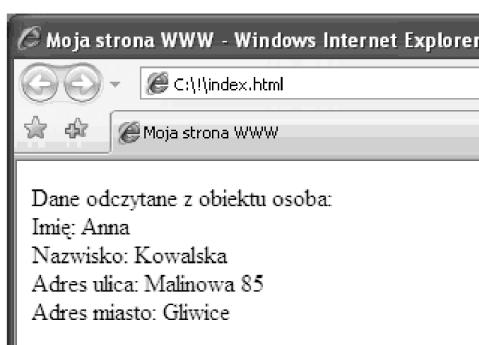
var str = "Dane odczytane z obiektu osoba:<br />";
str += "Imię: " + osoba.imię + "<br />";
str += "Nazwisko: " + osoba.nazwisko + "<br />";
str += "Adres ulica: " + osoba.adres.ulica + "<br />";
str += "Adres miasto: " + osoba.adres.miasto + "<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Definicja obiektu została przypisana zmiennej osoba. Jest to ta sama treść, co na listingu 3.6. Po wykonaniu takiej instrukcji na dostęp do obiektu pozwala zmenna osoba, a zatem wartość zapisaną we właściwości imię uzyskujemy przez odwołanie osoba. imię, nazwisko — osoba.nazwisko, ulica — osoba.adres.ulica, miasto — osoba.adres.miasto. Odczytywane dane są dopisywane do zmiennej str, dzięki czemu po wczytaniu kodu strony do przeglądarki zawartość obiektu zostanie wyświetlona na ekranie, co jest widoczne na rysunku 3.7.

Rysunek 3.7.

Dane odczytane
z obiektu zapisanego
w standardzie JSON



Bezpośrednie przypisywanie właściwości

JavaScript jest na tyle elastycznym językiem, że właściwości obiektu mogą być również przypisywane już po jego utworzeniu. Nie trzeba ich z góry definiować. Używamy do tego, podobnie jak przy odczytzie, operatora `.`(znak kropki). Po prostu, jeśli przy operacji przypisania:

```
obiekt.właściwość = wartość;
```

odwołamy się do nieistniejącej właściwości, zostanie ona utworzona. Możemy więc najpierw skonstruować pusty obiekt, a dopiero w dalszej części kodu przypisywać mu dane. Jak utworzyć pusty obiekt? Trzeba skorzystać ze składni z nawiasem klamrowym:

```
var obiekt = {};
```

Załóżmy, że chcielibyśmy przechowywać współrzędne punktu na płaszczyźnie, a więc `x` i `y`. Możemy najpierw utworzyć pusty obiekt punkt:

```
var punkt = {};
```

Następnie przypisać mu właściwość `x` o wartości 100:

```
punkt.x = 100;
```

A potem właściwość `y` o wartości 200:

```
punkt.y = 200;
```

Można je też później swobodnie odczytywać i modyfikować, co zostało zaprezentowane na listingu 3.8.

Listing 3.8. Bezpośrednie definiowanie właściwości

```
var punkt = {};
var str = "";

str += "Pierwotna zawartość obiektu punkt:<br />";
str += "x = " + punkt.x + "<br />";
str += "y = " + punkt.y + "<br />";

punkt.x = 100;
punkt.y = 200;

str += "Po zdefiniowaniu właściwości:<br />";
str += "x = " + punkt.x + "<br />";
str += "y = " + punkt.y + "<br />";

punkt.x = 0;
punkt.y = 0;

str += "Po zmianie właściwości:<br />";
str += "x = " + punkt.x + "<br />";
str += "y = " + punkt.y + "<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Najpierw został utworzony obiekt punkt, a następnie podjęta próba pobrania wartości właściwości *x* i *y*. Obiekt był jednak pusty (nie miał zdefiniowanych żadnych właściwości), zatem operacja nie mogła zakończyć się powodzeniem. Ponieważ próba odwołania do nieistniejącej właściwości daje w wyniku wartość undefined, takie też wartości zostały dopisane do zmiennej str przechowującej tekst komunikatów, które mają się pojawić na stronie WWW.

Dalej zostały wykonane instrukcje:

```
punkt.x = 100;
punkt.y = 200;
```

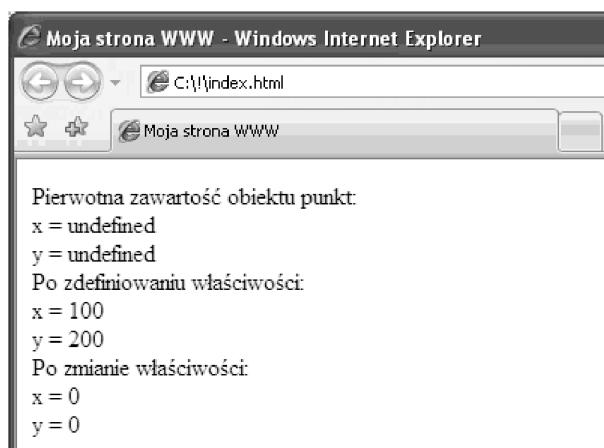
Jak już wiemy, tworzą one właściwości o nazwach *x* i *y* oraz przypisują im wartości 100 i 200. Skoro tak, można z powodzeniem odwołać się do tych właściwości i odczytać je. Do zmiennej str zostaną więc tym razem dopisane konkretne wartości właściwości *x* i *y*.

Wartość właściwości obiektu nie musi być stała w trakcie działania kodu. Dzięki temu w trzeciej części skryptu mogły zostać wykonane instrukcje:

```
punkt.x = 0;
punkt.y = 0;
```

Zmieniają one dotychczasowe wartości zapisane w *x* i *y* na 0. Ostatecznie, po uruchomieniu kodu w oknie przeglądarki zobaczymy widok zaprezentowany na rysunku 3.8.

Rysunek 3.8.
Wynik działania skryptu zmieniającego stan właściwości obiektu



Oprócz opisanego wyżej, istnieje jeszcze jeden sposób na dostęp do właściwości obiektu. Wykorzystano w nim składnię z nawiasem kwadratowym. Właściwość można zdefiniować lub uzyskać do niej dostęp za pomocą konstrukcji w postaci:

```
obiekt['nazwa_właściwości'] = wartość;
```

lub:

```
obiekt[identyfikator] = wartość;
```

W pierwszym przypadku nazwa właściwości podawana jest bezpośrednio, a w drugim korzystamy ze zmiennej pośredniczącej, która musi zawierać nazwę właściwości (ściślej — musi zawierać takie dane, aby mogły być skonwertowane na ciąg znaków, który zostanie użyty jako nazwa właściwości). Wszystkie podane sposoby definicji powodują utworzenie takich samych właściwości obiektów i można je stosować wymiennie. Spójrzmy na listing 3.9.

Listing 3.9. Różne sposoby dostępu do właściwości obiektów

```
var str = "";

var punkt = {}:
var osoba = {}:

punkt['x'] = 100;
punkt['y'] = 200;

var właściwość1 = "imię";
var właściwość2 = "nazwisko";

osoba[właściwość1] = "Anna";
osoba[właściwość2] = "Jabłońska";

str += "punkt.x = " + punkt.x;
str += "<br />";
str += "punkt['y'] = " + punkt['y'];
str += "<br />";
str += "osoba[właściwość1] = " + osoba[właściwość1];
str += "<br />";
str += "osoba['nazwisko'] = " + osoba['nazwisko'];

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Na początku zostały zadeklarowane dwa puste obiekty: punkt i osoba. Nie mają żadnych właściwości — są one przypisywane za pomocą oddzielnych instrukcji. Dla obiektu punkt są to:

```
punkt['x'] = 100;
punkt['y'] = 200;
```

Została zatem użyta pierwsza z omawianych składni. Nazwy właściwości przedstawiono jako ciągi znaków (umieszczonej je między znakami apostrofu⁶) i ujęto w nawias kwadratowy. Właściwości x została przypisana wartość 100, a y — 200. Mimo że wykorzystano inną technikę, powstał taki sam obiekt jak na początku listingu 3.8.

W przypadku obiektu osoba postapiliśmy inaczej. Najpierw powstały dwie zmienne, właściwość1 i właściwość2:

```
var właściwość1 = "imię";
var właściwość2 = "nazwisko";
```

⁶ Można też użyć znaków cudzysłowu — znaczenie będzie takie samo.

Zostały im przypisane ciągi znaków zawierające nazwy właściwości, które chcemy umieścić w obiekcie. Zmienne te zostały użyte do zdefiniowania właściwości:

```
osoba[właściwość1] = "Anna";
osoba[właściwość2] = "Jabłońska";
```

Jak rozumieć ten zapis? Ponieważ słowa w nawiasach kwadratowych nie zostały ujęte ani w znaki cudzysłowu, ani apostrofu, są traktowane jako identyfikatory — w tym przypadku nazwy zmiennych. A więc w miejsce ich wystąpienia są podstawiane ciągi znaków reprezentowane przez te zmienne, czyli oba powyższe zapisy będą potraktowane jako:

```
osoba["imię"] = "Anna";
osoba["nazwisko"] = "Jabłońska";
```

a więc w obiekcie osoba powstaną dwie właściwości o nazwach imię i nazwisko. Tym właściwościom zostaną przypisane ciągi znaków Anna i Jabłońska.

Niezależnie od sposobu, w jaki właściwości zostały zdefiniowane, można je odczytywać za pomocą dowolnej z przedstawionych do tej pory konstrukcji. Tak też dzieje się w przedstawionym skrypcie:

- ◆ właściwość x była definiowana jako punkt['x'], a jest odczytywana jako punkt.x;
- ◆ właściwość y była definiowana jako punkt['y'] i jest odczytywana jako punkt['y'];
- ◆ właściwość imię była definiowana jako osoba[właściwość1] i jest odczytywana jako osoba[właściwość1];
- ◆ właściwość nazwisko była definiowana jako osoba[właściwość2], a jest odczytywana jako osoba['nazwisko'];

Po co stosować konstrukcje z nawiasem kwadratowym? Czy nie wystarczy zwykła składnia, czyli *obiekt.właściwość*? Okazuje się, że nie zawsze. Spotkamy się bowiem z sytuacjami, gdy nazwa właściwości nie będzie z góry ustalona, ale będziemy ją otrzymywać z innego źródła. Może tak być, kiedy korzystamy z AJAX-a i nazwa właściwości jest przekazywana z serwera, ale też w dużo prostszych sytuacjach. Przykładowo użytkownik strony może wprowadzać pewne informacje w formularzu na stronie WWW, a my będziemy chcieli zapisać je jako właściwości obiektu. Wtedy nie obejdzie się bez użycia składni z nawiasem kwadratowym. Dane z formularza da się bowiem odczytać tylko jako串 znaków i nie uda się zastosować składni z kropką. Przyda się to również, gdy dane z obiektu będziemy odczytywać (lub zapisywać) automatycznie, za pomocą pętli. Przekonamy się o tym w kolejnej części lekcji.

Odczyt i zapis danych za pomocą pętli

Odczyt danych można przeprowadzić przy użyciu pętli. Można w tym celu posłużyć się specjalną wersją pętli for — pętlą for...in. Oto jej ogólna postać:

```
for (var nazwa in obiekt){
    //instrukcje
}
```

W każdym przebiegu pod zmienną nazwa zostanie podstawiona kolejna właściwość obiektu obiekt. Pętla będzie działała tak długo, aż odczytane zostaną wszystkie właściwości obiektu (lub też zostanie przerwana za pomocą instrukcji break lub return). Jest to bardzo wygodne, gdyż nie wymaga od nas ani znajomości nazw właściwości, ani też ich liczby.

Sposób użycia takiej pętli jest bardzo prosty. Założymy, że utworzyliśmy obiekt osoba zawierający pola: imię, nazwisko, rok_urodzenia, płeć:

```
var osoba = {  
    "imię": "Jan",  
    "nazwisko": "Nowak",  
    "rok_urodzenia": 1982,  
    "płeć": "M"  
}
```

Chcemy zastosować pętlę for...in, aby automatycznie wyświetlić nazwy właściwości i przypisane im dane. Użyjemy wtedy konstrukcji:

```
for(var indeks in osoba){  
    str += indeks + " = " + osoba[indeks] + "<br />";  
}
```

Zwróćmy uwagę, że używamy tu sposobu dostępu do właściwości wykorzystującego składnię z nawiasem kwadratowym. Nie można napisać np.:

```
osoba.indeks
```

gdyż oznaczałoby to, że chcemy otrzymać dostęp do właściwości o nazwie indeks, a takiej nie ma.

Dane można również odczytywać za pomocą pętli typu for, pod warunkiem że nazwy właściwości układają się w pewien schemat. Jeśli np. wiemy, że istnieje 5 właściwości o nazwach dana1, dana2 itd., użycie pętli for będzie jak najbardziej na miejscu. Co więcej, takie właściwości można również automatycznie zdefiniować. Jak to zrobić? Spójrzmy na skrypt z listingu 3.10.

Listing 3.10. Automatyczny zapis i odczyt właściwości

```
var str = "";  
  
var obiekt = {};  
var j = 10;  
  
for(i = 0; i < 5; i++){  
    obiekt["dana" + i] = j;  
    j += 10;  
}  
  
for(i = 0; i < 5; i++){  
    str += "dana" + i + " = ";  
    str += obiekt["dana" + i];  
    str += "<br />";  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Na początku został zdefiniowany pusty obiekt oraz zmienna pomocnicza *j* o wartości początkowej równej 10. Definicja właściwości odbywa się w pierwszej pętli for. Nazwy właściwości są tworzone według schematu *danaN*, gdzie *N* to wartości od 0 do 4, a więc *dana0*, *dana1* itd. Każdej właściwości przypisywana jest wartość zmiennej *j*, która zwiększa się o 10 w każdym przebiegu pętli (*j += 10;*), a więc *dana1 = 10*, *dana2 = 20* itd.

Odczyt danych odbywa się w drugiej pętli for. Ponieważ nazwy właściwości mają określony schemat, mogą być konstruowane przez połączenie ciągu *dana* ze stanem zmiennej iteracyjnej *i*. Odczytane wartości są dopisywane do zmiennej str:

```
str += obiekt["dana" + i];
```

dzięki czemu pojawią się na ekranie.

Funkcje jako właściwości obiektów

Właściwościami obiektów mogą być nie tylko zwykłe dane, takie jak liczby, ciągi znaków, inne obiekty itp., ale również funkcje. Takie funkcje nazywamy metodami. Jak połączyć funkcję z obiektem? Istnieje kilka możliwości. Na razie omówimy dwie z nich, a nieco bardziej zagłębimy się w tę tematykę w kolejnej lekcji. A zatem, jeżeli mamy gotowy pewien obiekt i chcielibyśmy dodać do niego funkcję (metodę), możemy postąpić według schematu:

```
obiekt.nazwa_funkcji = function(){
    // tutaj kod funkcji
}
```

Po takiej definicji funkcję można wywołać przy użyciu obiektu, w którym została umieszczona, stosując operator `:`

```
obiekt.nazwa_funkcji();
```

Taka funkcja zwykle wykonuje jakieś operacje na danych zawartych w obiekcie, musi więc mieć do nich dostęp. Używa się do tego konstrukcji ze słowem `this`:

```
this.nazwa_wlaściwości
```

To ważne, gdyż w ten sposób odróżnia się właściwości obiektu od innych zmiennych (lokalnych bądź globalnych). Najlepiej od razu wykonajmy prosty przykład. W poprzedniej części lekcji powstał m.in. obiekt punkt przechowujący współrzędne *x* i *y*. Warto by mieć metodę, która obliczy odległość takiego punktu od początku układu współrzędnych. Wzór obliczający tę odległość jest następujący⁷:

$$d = \sqrt{x^2 + y^2}$$

W JavaScriptie wyrazimy go tak:

```
d = Math.sqrt(x * x + y * y)
```

⁷ Zakładamy, że poruszamy się w układzie współrzędnych kartezjańskich.

lub tak:

```
d = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

Pierwsza postać jest czytelniejsza i szybsza, dlatego też z niej będziemy korzystać.

Gdybyśmy chcieli napisać zwykłą funkcję, która wykonuje takie zadanie, zapewne napisalibyśmy następujący kod:

```
function odległość(x, y)
{
    return Math.sqrt(x * x + y * y);
}
```

Jej przykładowe wywołanie miałyby postać:

```
var d = odległość(3, 4);
```

W przypadku funkcji przypisanej do obiektu cel jest nieco inny. Ma ona operować na właściwościach tego obiektu. Nie będzie więc przyjmowała argumentów, ale bezpośrednio odwoła się do odpowiednich danych. Przyjmie zatem postać:

```
function()
{
    return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

W praktyce będzie to wyglądało tak, jak na listingu 3.11.

Listing 3.11. Funkcja jako składowa obiektu

```
var str = "";
var punkt = {};
punkt.x = 3;
punkt.y = 4;

punkt.odległość = function()
{
    return Math.sqrt(this.x * this.x + this.y * this.y);
}

var d = punkt.odległość();

str += "Wynikiem wywołania metody odległość obiektu punkt";
str += " jest wartość " + d + ".";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Na początku został zdefiniowany pusty obiekt punkt, a następnie zostały mu przypisane właściwości x i y o wartościach 3 i 4. Tego typu konstrukcje tworzyliśmy już we wcześniejszej części lekcji. Dalej znajduje się jednak definicja funkcji przypisanej jako właściwość obiektu punkt. Właściwość ta ma nazwę odległość, a ponieważ jest funkcją, możemy ją określić jako metodę obiektu punkt. Metoda ta za pomocą instrukcji

return zwraca wynik obliczeń zgodnych z podanym wcześniej wzorem. Ponieważ opiera się na wartościach obiektu punkt, odwołuje się do nich za pomocą składni ze słowem this. Słowo this oznacza w tej sytuacji prostu bieżący obiekt (odwołanie do bieżącego obiektu). A zatem pod this.x zostanie podstawiona wartość 3, a pod this.y — wartość 4. Wynikiem działania metody będzie więc 5 (bowiem $\sqrt{3^2 + 4^2} = 5$). Przekonujemy się o tym, wywołując metodę odległość obiektu punkt w instrukcji:

```
var d = punkt.odleglosc();
```

Wywołanie metody jest tym samym, co wywołanie funkcji (wykonywane są zwarte w niej instrukcje), z tą różnicą, że metodę wywołujemy na rzecz obiektu lub inaczej — poprzez obiekt, w którym jest zawarta.

We wnętrzu metody możemy wykonywać takie same operacje jak w funkcjach. Można w nich korzystać ze zmiennych globalnych i lokalnych, zmienne te mogą mieć też takie same nazwy jak właściwości obiektu, w którym znajduje się dana metoda. Jest to możliwe, bowiem — jak już wiemy — dostęp do właściwości odbywa się przy użyciu składni ze słowem this. Gdybyśmy zatem chcieli w obliczeniach skorzystać z dodatkowych zmiennych pomocniczych, metoda odległość z listingu 3.11 mogłaby przyjąć postać:

```
punkt.odleglosc = function()
{
    var x = this.x;
    var y = this.y;
    return Math.sqrt(x * x + y * y);
}
```

Została tu utworzona zmienna lokalna funkcji odległość o nazwie x (var x), której przypisano wartość właściwości x obiektu punkt (this.x). Analogiczna operacja została wykonana w stosunku do właściwości y (this.y) — jej wartość została przypisana lokalnej zmiennej y (var y). Następnie na zmiennych lokalnych zostały wykonane obliczenia, a ich wynik zwrocony za pomocą instrukcji return.

Co jednak zrobić, jeśli chcemy od razu zdefiniować cały obiekt w formacie JSON? Nie ma problemu. Oprócz wymienionych w punkcie „Tworzenie prostych obiektów” składowych obiektu, mogą wystąpić również funkcje. Ogólna definicja ma wtedy postać:

```
{
    definicje właściwości,
    nazwa_funkcji : function()
    {
        //definicja funkcji
    }
}
```

Za pomocą tej konstrukcji bez problemów utworzymy obiekt zawierający od razu zarówno dane, jak i funkcje. Definicja obiektu punkt zawierającego metodę odległość mogłaby zatem przyjąć również postać widoczną na listingu 3.12. Taki obiekt będzie można zastosować w dokładnie taki sam sposób, jak ten z listingu 3.11.

Listing 3.12. Funkcja jako składowa obiektu w standardzie JSON

```
var punkt = {  
    x : 3,  
    y : 4,  
    odległość : function()  
    {  
        return Math.sqrt(this.x * this.x + this.y * this.y);  
    }  
};
```

Funkcje będące składowymi obiektu mogą również przyjmować argumenty. Definiuje się je, podobnie jak w przypadku zwyczajnych funkcji, podając nazwy argumentów w nawiasie okrągłym. Różnica jest taka, że w metodach nawias ten występuje za słowem `function`. Jeżeli więc chcemy utworzyć obiekt `punkt`, zawierający:

- ◆ pole `x` — reprezentujące współrzędną `x`;
 - ◆ pole `y` — reprezentujące współrzędną `y`;
 - ◆ metodę `odległość` — zwracającą odległość punktu od początku układu współrzędnych;
 - ◆ metodę `przesuń` — przesuwającą punkt o zadaną wartość w pionie i poziomie;
- możemy zrobić to tak, jak zaprezentowano na listingu 3.13.

Listing 3.13. Użycie metody przyjmującej argumenty

```
var str = "";  
  
var punkt = {  
    x : 3,  
    y : 4,  
    odległość : function()  
    {  
        return Math.sqrt(this.x * this.x + this.y * this.y);  
    },  
    przesuń : function(px, py)  
    {  
        this.x += px;  
        this.y += py;  
    }  
};  
  
var d1 = punkt.odległość();  
punkt.przesuń(2, 6);  
var d2 = punkt.odległość();  
  
str += "Początkowa odległość punktu od początku ";  
str += "układu współrzędnych to " + d1 + ".<br />";  
str += "Po przesunięciu odległość punktu od początku ";  
str += "układu współrzędnych to " + d2 + ".<br />";  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Cały obiekt został zdefiniowany za pomocą składni w standardzie JSON. Znajdują się w nim dwa pola, `x` i `y`, oraz dwie metody, czyli odległość i przesuń. Pierwsza z metod działa tak samo jak wcześniej przedstawiona i jest bezargumentowa. Metoda przesuń przyjmuje natomiast dwa argumenty: `px` i `py`. Argument `px` oznacza przesunięcie wzdłuż osi OX, a `py` — wzdłuż osi OY. To oznacza, że wartość właściwości `x` ma zostać zwiększone o `px`:

```
this.x += px;
```

a właściwości `y` — o `py`:

```
this.y += py;
```

Za definicję obiektu znajdują się instrukcje testujące jego działanie. Najpierw jest pobierana i zapisywana w zmiennej `d1` początkowa odległość punktu reprezentowanego przez punkt od początku układu współrzędnych:

```
var d1 = punkt.odleglosc();
```

Następnie za pomocą metody przesuń punkt jest przesuwany (zmieniane są jego współrzędne):

```
punkt.przesun(2, 6);
```

I pobierana jest jego aktualna odległość od początku układu:

```
var d2 = punkt.odleglosc();
```

Uzyskane wartości są dopisywane do zmiennej `str`, a tym samym, wyświetlane na ekranie.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 8.1.

Zmodyfikuj kod z listingu 3.7 tak, aby dane dotyczące imienia i nazwiska w obiekcie osoba tworzyły osobny obiekt.

Ćwiczenie 8.2.

Zmień kod z listingu 3.10 tak, aby odczyt danych z obiektu odbywał się w pętli `for...in`.

Ćwiczenie 8.3.

Utwórz obiekt zawierający 10 właściwości o nazwach zgodnych ze schematem LN , gdzie N to indeks od 1 do 10, a L to litera A dla indeksów parzystych i B dla nieparzystych (czyli nazwy kolejnych właściwości to B1, A2, B3, A4 itd.). Właściwościom o indeksach parzystych przypisz wartość 0, a pozostałym — 1. Do utworzenia obiektu użyj pętli.

Ćwiczenie 8.4.

Do funkcji przesuń z listingu 3.13 dodaj instrukcje sprawdzające, czy argumenty mogą być interpretowane jako wartości liczbowe. Jeśli nie mogą, nie dokonuj modyfikacji właściwości `x` i `y`.

Ćwiczenie 8.5.

Utwórz obiekt reprezentujący prostokąt na płaszczyźnie. Zawrzyj w nim metodę zwierającą pole tego prostokąta.

Lekcja 9. Funkcje, konstruktory i prototypy

W lekcji 8. poznaliśmy pojęcie obiektu. Wiemy już, jak go tworzyć i posługiwać się właściwościami oraz metodami. W lekcji 9. będziemy kontynuować ten temat. Poznamy nowe, bardziej zaawansowane, ale też bardzo wygodne sposoby tworzenia tych konstrukcji programistycznych. Zdecydowanie rozszerzymy też wiadomości na temat poznanych w lekcji 6. funkcji. Dowiemy się także, czym są konstruktory i prototypy, oraz poznamy sposób, w jaki wspomagają programowanie z użyciem obiektów.

Czy funkcje to obiekty?

W lekcji 6. w rozdziale 2. poznaliśmy pojęcie funkcji. W lekcji 7. okazało się, że funkcję można przypisać jako właściwość obiektu. Co więcej, przypisanie funkcji czy nawet definicja w formacie JSON wyglądały tak samo jak przypisanie zwykłej wartości (np. liczby czy ciągu znaków). Czy nie wydawało się to trochę dziwne? A skoro można przypisać funkcję do właściwości obiektu, to może można przypisać ją również do zwykłej zmiennej? Okazuje się, że tak. W pełni prawidłowa będzie konstrukcja o schematycznej postaci:

```
var zmienna = function(argumenty_funkcji){  
    //treść funkcji  
}
```

Jeśli np. chcemy zmiennej `razyDwa` przypisać funkcję, która zwraca podwojoną wartość argumentu, zastosujemy instrukcję:

```
var razyDwa = function(arg){  
    return 2 * arg;  
}
```

Jak wywołać (uruchomić) taką funkcję? Po prostu, używając nawiasu okrągłego:

```
razyDwa(8);
```

Instrukcja ta powoduje wywołanie funkcji przypisanej zmiennej razyDwa i przekazanie jej liczby 8 w postaci argumentu. Co więcej, wartość zmiennej razyDwa można przypisać innej zmiennej. Spójrzmy na listing 3.14.

Listing 3.14. Funkcja przypisywana zmiennej

```
var str = "";

var razyDwa = function(arg){
    return 2 * arg;
}

var pomnóż = razyDwa;

var wynik1 = razyDwa(8);
var wynik2 = pomnóż(8);

str += "razyDwa(8) = " + wynik1;
str += "<br />";
str += "pomnóż(8) = " + wynik2;

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Funkcja przyjmująca jeden argument i zwracająca jego podwojoną wartość została przypisana zmiennej razyDwa. Następnie powstała zmienna pomnóż i została jej przypisana wartość zmiennej razyDwa:

```
var pomnóż = razyDwa;
```

To oznacza, że zarówno razyDwa, jak i pomnóż wskazują tę samą funkcję i można ich używać do jej wywoływania. Tak też stało się w dalszej części kodu. Zmiennej wynik1 została przypisany wynik wywołania razyDwa(8), a zmiennej wynik2 — wynik wywołania pomnóż(8). Oczywiście, wyniki będą takie same, bowiem wywołana została ta sama funkcja z takim samym argumentem (wartość 8).

Skoro funkcja może być przypisana zmiennej, może być również przekazana jako argument innej funkcji. Nic przecież nie stoi na przeszkodzie, aby jako argumentu użyć utworzonej przez nas zmiennej razyDwa czy pomnóż. Funkcja przekazana jako argument może też być wywołana bez żadnych problemów. Rozważmy przykład widoczny na listingu 3.15.

Listing 3.15. Funkcja jako argument innej funkcji

```
var f1 = function(arg){
    arg();
}

var f2 = function(){
    alert("Jestem funkcją f2.");
}

f1(f2);
```

Mamy tutaj funkcję przyjmującą jeden argument (arg), przypisaną zmiennej f1. W jej wnętrzu znajduje się instrukcja:

```
arg();
```

Wiemy, że w normalnej sytuacji oznaczałaby ona wywołanie funkcji o nazwie arg. Ponieważ jednak arg jest nazwą argumentu, instrukcja oznacza wywołanie funkcji przekazanej jako argument. Oczywiście, aby taka konstrukcja mogła zostać wykonana prawidłowo, argumentem przekazanym funkcji przypisanej zmiennej f1 musi być inna funkcja — inaczej wystąpi błąd.

Druga z funkcji została przypisana zmiennej f2. Ma zwyczajną postać. W jej wnętrzu znajduje się instrukcja alert wyświetlająca w oknie dialogowym pewien komunikat.

Na samym końcu kodu znajduje się instrukcja:

```
f1(f2);
```

Jest to więc wywołanie funkcji f1 i przekazanie jej w postaci argumentu zmiennej f2, czyli funkcji przypisanej tej zmiennej. Znaczy to, że funkcja f1 otrzymuje jako argument również funkcję i wywołuje ją w swoim wnętrzu (instrukcja arg()). Na ekranie pojawi się więc okno dialogowe z napisem Jestem funkcją f2.

Powyższy przykład jest jednak bardzo teoretyczny. Pokazuje jedynie strukturę kodu pozwalającego użyć funkcji jako argumentu. Wbrew pozorom jednak, tego typu konstrukcje wcale nie są jedynie hipotetyczną możliwością, ale są używane w praktyce. Co prawda, korzysta się z nich zazwyczaj w bardziej zaawansowanych skryptach, wykonajmy jednak bardzo uproszczony przykład, pokazujący, do czego mogą się przydać. Zapisany został na listingu 3.16.

Listing 3.16. Praktyczne użycie funkcji jako argumentu

```
var suma = function(arg1, arg2){  
    return arg1 + arg2;  
}  
  
var różnica = function(arg1, arg2){  
    return arg1 - arg2;  
}  
  
function oblicz(arg1, arg2, działanie){  
    var wynik = działanie(arg1, arg2);  
    return Math.sqrt(wynik);  
}  
  
var wynik1 = oblicz(10, 5, suma);  
var wynik2 = oblicz(10, 5, różnica);  
  
alert('Pierwszy wynik to ' + wynik1);  
alert('Drugi wynik to ' + wynik2);
```

Zmiennej suma przypisana została funkcja przyjmująca dwa argumenty i zwracająca ich sumę, a zmiennej różnica funkcja działająca analogicznie, ale zwracająca różnicę. Funkcja oblicz została zdefiniowana w sposób klasyczny i przyjmuje trzy argumenty. Dwa pierwsze (arg1 i arg2) określają argumenty, na których będą wykonywane działania, a trzeci (działanie) — funkcję, która wykona na tych argumentach dodatkową operację. Funkcja oblicz najpierw wykonuje na przekazanych jej argumentach pewną operację, wykorzystując do tego inną funkcję, przekazaną jej jako trzeci argument wywołania:

```
var wynik = działanie(arg1, arg2);
```

A następnie wynik tej operacji poddaje działaniu metody sqrt i za pomocą instrukcji return zwraca ostateczny rezultat. W celu przetestowania poprawności kodu zostały też wykonane instrukcje:

```
var wynik1 = oblicz(10, 5, suma);
var wynik2 = oblicz(10, 5, różnica);
```

Ich wyniki wyświetlono w dwóch oknach dialogowych.

Łatwo jednak zauważyc, że wszystkie działania mogły być wykonane bezpośrednio w funkcji oblicz, a trzeci argument mógłby być po prostu znakiem określającym rodzaj działania. Co zatem zyskaliśmy, stosując taką dla osób początkujących zapewne dużo bardziej skomplikowaną konstrukcję? Otóż, dużo większą elastyczność takiego rozwiązania. Przy użyciu takiego podejścia w funkcji oblicz można zatrzymać jedynie niezmienną na pewno część kodu, która nie będzie modyfikowana, a sterować jej zachowaniem z zewnątrz, przekazując odpowiednie funkcje. To nie musi być przecież tylko suma, różnica i pierwiastek, obliczenia mogą być o wiele bardziej zaawansowane, ale zasada pozostaje niezmienna. Tego typu rozwiązania stosuje się często, gdy nad projektem pracuje więcej osób lub też korzystamy z zestawów skryptów (bibliotek) pobranych zewnętrznych źródeł (np. z internetu). A że ma to bardzo praktyczne znaczenie, przekonamy się już w lekcji 10., gdy zajmiemy się sortowaniem tablic.

Wciąż jednak bez odpowiedzi pozostaje pytanie zawarte w tytule tej części lekcji. Z pewnością domyślamy się, że ta odpowiedź jest twierdząca. Tak, funkcje są obiektami. Pisząc:

```
function (argumenty){/*treść funkcji*/}
```

definiujemy obiekt funkcji. To dlatego mogliśmy przypisać taką definicję zmiennej. De facto taka zmienna wskazuje po prostu obiekt funkcji. Ważne, abyśmy pamiętali, że ten obiekt powstaje niezależnie od sposobu, w jaki utworzymy funkcję. Sposób poznany w lekcji 6. jest w pełni równoważny z tymi, które podano w lekcji bieżącej.

Choć może być to zaskakujące, definicja w postaci:

```
var nazwa_funkcji = function(argumenty)
{
    // treść funkcji
}
```

jest w praktyce tym samym, co zapis:

```
function nazwa_funkcji (argumenty)
{
    //treść funkcji
}
```

W obu przypadkach powstał obiekt funkcji (nie ma on nazwy!) i został przypisany zmiennej *nazwa_funkcji*. To za pomocą tej zmiennej wywołujemy funkcję.

Właściwości funkcji

Skoro funkcje są obiektami, mogą również mieć własne właściwości. Taką właściwością jest `arguments`. Jest to obiekt (podobny w zachowaniu do tablicy) przechowujący listę argumentów, z jakimi dana funkcja została wywołana. Przypomnijmy sobie materiał z lekcji 6. Zgodnie z przedstawionymi w niej informacjami, można pominąć w wywołaniu funkcji niektóre z jej argumentów. Rzeczywistość jest jednak jeszcze ciekawsza. Otóż, funkcje można wywołać, przekazując więcej argumentów niż w niej zadeklarowano. Wtedy jednak nie mamy do nich bezpośredniego dostępu i musimy skorzystać z obiektu pośredniczącego `arguments`. Dostęp do pierwszego argumentu osiągniemy przy użyciu odwołania `arguments[0]`, do drugiego — `arguments[1]` itd. Zobrazowano to w przykładzie widocznym na listingu 3.17.

Listing 3.17. Dostęp do argumentów funkcji

```
function f(arg1, arg2){
    var str = "";
    str += "wartość arg1 = " + arg1;
    str += "\nwartość arg2 = " + arg2;
    str += "\nwartość arguments[0] = " + arguments[0];
    str += "\nwartość arguments[1] = " + arguments[1];
    str += "\nwartość arguments[2] = " + arguments[2];
    alert(str);
}

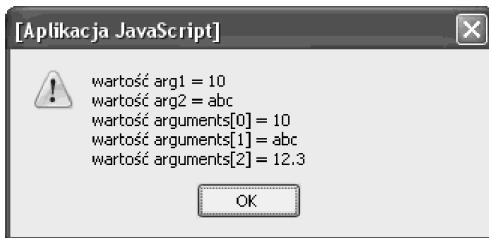
f(10, "abc", 12.3);
```

Funkcja `f` została zadeklarowana z dwoma argumentami, `arg1` i `arg2`, ale wywołana z trzema:

```
f(10, "abc", 12.3);
```

Dostęp do pierwszego można uzyskać w sposób klasyczny, pisząc `arg1` lub też korzystając z konstrukcji `arguments[0]`. Podobnie jest z argumentem drugim, bo albo piszemy `arg2`, albo `arguments[1]`. Z trzecim jest inaczej. Ponieważ nie został zadeklarowany w definicji funkcji, trzeba użyć konstrukcji `arguments[2]`, aby otrzymać do niego dostęp i odczytać zawartość. W funkcji `f` do odczytania wartości argumentów zostały użyte wszystkie wymienione sposoby, a więc po uruchomieniu skryptu na ekranie zobaczymy okno dialogowe zaprezentowane na rysunku 3.9.

Rysunek 3.9.
Różne sposoby
odczytywania
argumentów



W powyższym przykładzie odczytanie trzeciego argumentu odbyło się bez problemów, bowiem z góry założyliśmy, że funkcja zostanie wywołana dokładnie z trzema argumentami. W praktyce nie będzie to takie oczywiste. Równie dobrze mógłby to być tylko jeden argument albo cztery czy pięć. Jeżeli zatem w danej funkcji oczekujemy pewnej dokładnej liczby argumentów i nie może być ona wywołana z inną ich liczbą, trzeba w jakiś sposób zbadać, ile ich jest. Umożliwia to właściwość `length` obiektu `arguments`. Zawiera ona wartość określającą dokładną liczbę argumentów funkcji. Możemy to wykorzystać w sposób zaprezentowany na listingu 3.18.

Listing 3.18. Badanie liczby argumentów

```
function suma(arg1, arg2){
    if(arguments.length != 2){
        var str = "Błędna liczba argumentów.\n";
        str += "Oczekiwano 2, otrzymano ";
        str += arguments.length + ".";
        alert(str);
        return NaN;
    }
    return arg1 + arg2;
}

suma(10, 20);
suma(1, 2, 3);
```

Funkcja `suma` przyjmuje dwa argumenty i ma zwrócić ich sumę. Ma się tak stać tylko wtedy, jeżeli zostanie wywołana dokładnie z dwoma argumentami, każda inna ich liczba ma spowodować wyświetlenie komunikatu o błędzie i zwrócenie wartości `NaN`. Za pomocą instrukcji warunkowej `if` badamy zatem, czy właściwość `arguments.length` jest różna od dwóch:

```
if(arguments.length != 2){
```

Jeśli tak jest, konstruujemy komunikat zawierający informację o oczekiwanej i faktycznie przekazanej liczbie argumentów, wyświetlamy go na ekranie za pomocą instrukcji `alert` i za pomocą instrukcji `return` zwracamy wartość `NaN`. W przeciwnym przypadku zwracamy sumę argumentów (`return arg1 + arg2;`).

W celach testowych funkcja `suma` została wywołana dwa razy. Pierwszy raz z dwoma argumentami, a drugi — z trzema. Dzięki temu możemy przekonać się, że działa zgodnie z założeniami.

Przy stosowaniu odwołań do właściwości arguments musimy pamiętać, że może ona zostać przesłonięta przez argument lub zmienną lokalną o takiej samej nazwie. Aby uniknąć niejednoznaczności, lepiej więc nie używać słowa arguments jako identyfikatora. Przykładowo konstrukcje:

```
function f(arguments, arguments2)
{
    //treść funkcji
}
```

lub:

```
function f(arg1, arg2)
{
    var arguments = 123;
    //treść funkcji
}
```

są formalnie prawidłowe, ale uniemożliwiają odwołanie się do obiektu arguments przechowującego argumenty funkcji. W pierwszym przypadku będzie tak dlatego, że słowo arguments zostało użyte jako nazwa argumentu, a w drugim — ponieważ słowo to zostało użyte jako nazwa zmiennej.

Oprócz właściwości length, obiekt arguments zawiera również właściwość o nazwie callee. Jest to odniesienie do aktualnej funkcji, czyli jeśli mamy funkcję o przykładowej nazwie suma, to w jej wnętrzu odwołanie argumentscallee jest odwołaniem do niej samej (do funkcji suma). Nie będziemy z tej właściwości korzystać, ale warto wiedzieć, że istnieje. Umożliwia m.in. rekurencyjne wywołania funkcji anonimowych (nieposiadających nazwy).

Obiekt globalny

Interpreter JavaScriptu w trakcie swojego uruchamiania (co będzie równoznaczne ze startem przeglądarki, bowiem tylko takie sytuacje omawiamy w tej książce) tworzy tzw. obiekt globalny (z ang. *global object*). To obiekt nadrzędny dla wszystkich innych. Jest odpowiednio inicjalizowany, m.in. tworzone są jego właściwości i metody. Są to opisane w lekcji 7. funkcje i właściwości globalne. Zatem nie są one osobnymi i niezależnymi tworami, ale składowymi obiektu globalnego. Przykładowo funkcja parseInt to metoda obiektu globalnego, a NaN jest jego właściwością. Do tego obiektu możemy się odwołać przy użyciu słowa this, choć nie jest to konieczne. Skrypt jest bowiem uruchamiany w kontekście tego obiektu (w uproszczeniu można by powiedzieć, że wewnątrz obiektu), a więc automatycznie uzyskuje dostęp do jego składowych. Niemniej prawidłowe są również odwołania typu:

```
this.parseInt("ciąg znaków")
```

lub:

```
this.Math.sqrt(4);
```

Zauważmy przy tym, że — zgodnie z powyższą składnią — Math to obiekt będący właściwością obiektu this (globalnego), a sqrt to metoda (funkcja) obiektu Math.

Co więcej, każda zmienna globalna, a także funkcja globalna (czyli zdefiniowana na najwyższym poziomie kodu skryptu, niezagnieżdżona w innej funkcji i niebędąca składową jakiegoś obiektu) są w istocie właściwościami obiektu globalnego.

Jeżeli zatem w skrypcie napiszemy:

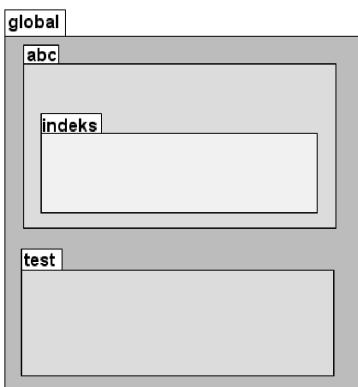
```
var abc = 123;
function test()
{
    // treść funkcji
}
```

oznacza to definicję właściwości abc oraz metody test obiektu globalnego. Obie te właściwości są też obiektami! To, że funkcja jest obiektem, już wiemy. Okazuje się jednak, że zwykła zmienna typu prostego to też obiekt. Bez problemu można np. zdefiniować nową właściwość tego obiektu. Wystarczy do powyższego kodu dodać przykładową instrukcję:

```
abc.indeks = 100;
```

Do zmiennej (obiektu) abc dodamy wtedy nową właściwość o nazwie indeks i wartość 100. Sytuacja będzie wtedy wyglądała tak, jak na rysunku 3.10.

Rysunek 3.10.
Schemat zagnieżdzania
obiektów



Każdy uważny czytelnik na pewno zada teraz pytanie: „Jak w takim razie traktować zmienne lokalne funkcji, a także funkcje wewnętrzne (zagnieżdzone)?”. Odpowiedź nie jest skomplikowana. Skoro funkcja jest obiektem, jej zmienne lokalne (zdefiniowane za pomocą słowa var wewnątrz funkcji) są właściwościami tego obiektu (obiektami zawartymi w obiekcie). Podobnie z funkcjami zagnieżdżonymi — to obiekty będące właściwościami obiektu funkcji zewnętrznej. Przykładowy fragment kodu:

```
function zewnętrzna()
{
    function wewnętrzna()
    {
        var def = 456;
        // dalsza część kodu
    }
    var abc = 123;
    // dalsza część kodu
}
```

można by schematycznie reprezentować tak, jak zostało to pokazane na rysunku 3.11.

Rysunek 3.11.
*Obiektowy schemat
zagnieżdżania funkcji*



Konstruktory

Oprócz przedstawionych do tej pory sposobów tworzenia obiektów, istnieje jeszcze jeden. Jest nim użycie operatora new. W najprostszym przypadku będzie miał postać:

```
var obiekt = new Object();
```

Powstanie w ten sposób nowy pusty obiekt, który zostanie przypisany zmiennej obiekt. Jest to więc odpowiednik konstrukcji:

```
var obiekt = {};
```

Takiemu pustemu obiektowi można przypisywać właściwości i metody, tak jak odbywało się to w dotychczasowych przykładach, np.:

```
var obiekt = new Object();
obiekt.x = 100;
```

Konstrukcje typu new Object() są jednak dużo bardziej użyteczne. Co bowiem przy pomina wyrażenie Object()? Czyż nie jest podobne do wywołania funkcji? Oczywiście! Nie tylko jest podobne, ale też stanowi wywołanie funkcji. Funkcja ta jest po prostu argumentem operatora new. Działanie operatora new składa się z dwóch kroków. Najpierw tworzy nowy, pusty obiekt, a następnie przekazuje go funkcji, będącej jego argumentem, jako wskazanie this. Zatem w funkcji, która jest argumentem operatora new, słowo this wskazuje nowo utworzony obiekt. Zadaniem tej funkcji jest zainicjalizowanie tego obiektu, skonstruowanie go, dlatego też nazywamy ją konstruktorem (z ang. *constructor*). Taki konstruktor możemy napisać sami, a tym samym możemy tworzyć własne obiekty.

Jest to bardzo użyteczne. Przypomnijmy sobie, jak w lekcji 8. tworzyliśmy obiekty typu punkt. Najpierw budowaliśmy pusty obiekt:

```
var punkt = {};
```

Następnie jego właściwości:

```
punkt.x = 0;  
punkt.y = 0;
```

Za każdym razem, gdy chcieliśmy skorzystać z punktu, trzeba było powtarzać te operacje. A co by się stało, gdyby właściwości było więcej, 10, 20, 100? Definiowanie 100 właściwości w każdym miejscu, w którym chcielibyśmy skorzystać z danego obiektu, byłoby bardzo niewygodne. Lepiej zlecić to funkcji będącej konstruktorem, który mógłby wyglądać tak:

```
function Punkt()  
{  
    this.x = 0;  
    this.y = 0;  
}
```

Po takiej definicji nowy obiekt typu Punkt utworzymy, pisząc:

```
new Punkt();
```

Oto przykład:

```
var punkt = new Punkt();
```

W ten sposób powstał obiekt przypisany zmiennej punkt, posiadający dwie właściwości, x i y, zainicjalizowane wartością 0. Można powiedzieć, że tworząc funkcję konstruktora, zbudowaliśmy nowy typ danych o nazwie Punkt, który reprezentuje współrzędne punktów na płaszczyźnie.

Konstruktor, jak każda inna funkcja, może przyjmować argumenty. W naszym przykładzie byłoby dobrze, aby funkcja Punkt przyjmowała argumenty określające początkowe wartości współrzędnych x i y. Miałaby ona wtedy następującą postać:

```
function Punkt(x, y)  
{  
    this.x = x;  
    this.y = x;  
}
```

Wtedy, gdy zechcemy przypisać początkowe współrzędne x i y równe 100 i 200, napiszemy:

```
var punkt = new Punkt(100, 200);
```

Oczywiście, początkowe wartości tak utworzonego obiektu mogą być dowolnie odczytywane bądź zmieniane i będzie się on zachowywał tak samo jak obiekty tworzone za pomocą innych konstrukcji. Na listingu 3.19 jest widoczny skrypt korzystający z nowo poznanej konstrukcji.

Listing 3.19. Tworzenie obiektów z użyciem konstruktorów

```
var str = "";
function Punkt(x, y)
{
    this.x = x;
    this.y = y;
}

function getXYStr()
{
    var str = "";
    str += "punkt1.x = " + punkt1.x + "<br />";
    str += "punkt1.y = " + punkt1.y + "<br /><br />";
    str += "punkt2.x = " + punkt2.x + "<br />";
    str += "punkt2.y = " + punkt2.y + "<br /><br />";
    return str;
}

var punkt1 = new Punkt(1, 10);
var punkt2 = new Punkt(20, 30);

str += "Pierwotne wartości obiektów:<br />";
str += getXYStr();

punkt1.x = 50;
punkt2.y = 8;

str += "Wartości obiektów po zmianie:<br />";
str += getXYStr();

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Funkcja konstruktora ma tu postać przedstawioną wyżej. Przyjmuje dwa argumenty i przypisuje ich wartości polom x i y obiektu wskazywanego przez `this`. A zatem nowo utworzony obiekt otrzymuje pola x i y o wartościach wskazywanych przez argumenty x i y .

Za konstruktorem znajduje się funkcja pomocnicza `getXYStr`. Zwraca ona ciąg opisujący wartości pól obiektów globalnych `punkt1` i `punkt2`. A więc odczytuje ich właściwości, buduje komunikat informujący o wartościach tych właściwości i zwraca go jako rezultat swojego działania. Dzięki temu w dalszej części kodu nie musimy wielokrotnie powtarzać tych samych instrukcji.

Za definicjami funkcji tworzone są dwa obiekty typu `Punkt`, czyli `punkt1` i `punkt2`. Pierwszy o współrzędnych 1 i 10 oraz drugi o współrzędnych 20 i 30. Wartości pól utworzonych w ten sposób obiektów są pobierane za pomocą funkcji `getXYStr` i dopisywane do zmiennej `str`. Następnie modyfikowana jest współrzędna x obiektu `punkt1` i współrzędna y obiektu `punkt2`. Wartości obiektów po modyfikacji są również pobierane za pomocą funkcji `getXYStr`.

Zwrócićmy przy tym uwagę, że funkcja konstruktora nie zwraca żadnego wyniku, ope-ruje tylko na obiekcie utworzonym i przekazanym jej przez operator new. Obiekt ten jest wartością wyrażenia new *Funkcja()*. Zatem najpierw tworzony jest nowy, pusty obiekt, potem wywoływana funkcja, która ma do niego dostęp poprzez wskazanie this, i na zakończenie obiekt ten staje się wartością całego wyrażenia. Dlatego też można napisać:

```
var zmienna = new Funkcja();
```

Jeśli jednak takie działanie nam nie odpowiada i z pewnych względów sami chcemy utworzyć obiekt, również możemy to zrobić. Wtedy, po samodzielnym utworzeniu obiektu należy zwrócić go jako rezultat działania konstruktora. Schemat takiego konstruktora wygląda następująco:

```
function Konstruktor(argumenty)
{
    // tutaj utwórz nowy obiekt, np.:
    var obj = new Object();
    // tutaj utwórz składowe obiektu obj,
    // wykonaj inne czynności konstruktora
    return obj;
}
```

Taka konstrukcja jest rzadziej spotykana. Przy tworzeniu obiektu za pomocą tego typu konstruktora, czyli po zastosowaniu instrukcji:

```
var obiekt = new Konstruktor(argumenty);
```

postępowanie interpretera jest następujące:

- ◆ tworzony jest nowy, pusty obiekt i przekazywany funkcji *Konstruktor*, gdzie jest dostępny w postaci wskazania this,
- ◆ wywoływana jest funkcja *Konstruktor*, która wykonuje zawarte w niej instrukcje,
- ◆ ponieważ funkcja *Konstruktor* w rezultacie swojego działania zwróciła wartość, pierwotny obiekt (ten ze wskazania this) jest usuwany,
- ◆ rezultatem wyrażenia new staje się obiekt zwrócony przez funkcję *Konstruktor*,
- ◆ zwrócony obiekt jest przypisywany zmiennej obiekt.

Prototypy

W lekcji 8., korzystając ze składni JSON, tworzyliśmy obiekty punkt zawierające metody odległość i przesun. Jednak, podobnie jak ma to miejsce w przypadku innych właściwości, przypisywanie metody w trakcie tworzenia każdego obiektu byłoby bardzo niewygodne i nieefektywne. Rozwiążaniem, które zapewne od razu się nasuwa, jest zdefiniowanie metod w konstruktorze. Skoro bowiem można w nim umieścić pola obiektu, to na pewno uda się to zrobić również z metodami. Ogólna definicja takiego konstruktora miałaby wtedy postać:

```
function Konstruktor(argumenty)
{
    // tutaj utwórz pola obiektu
    this.metoda = function(argumenty)
    {
        // treść metody
    }
}
```

Dodajmy więc do funkcji tworzącej obiekty typu Punkt definicję metody odległość i spróbujmy jej użyć w przykładowym skrypcie. Można to zrobić tak, jak przedstawiono na listingu 3.20.

Listing 3.20. Tworzenie metod w konstruktorze

```
function Punkt(x, y)
{
    this.x = x;
    this.y = y;
    this.odległość = function()
    {
        return Math.sqrt(
            this.x * this.x + this.y * this.y);
    }
}

var punkt1 = new Punkt(1, 10);
var punkt2 = new Punkt(20, 30);
var d1 = punkt1.odległość();
var d2 = punkt2.odległość();

alert("punkt1.odległość = " + d1);
alert("punkt2.odległość = " + d2);
```

Konstruktor jest podobny do tego z listingu 3.19. Przyjmuje dwa argumenty określające współrzędne *x* oraz *y* i przypisuje je właściwościom *this.x* i *this.y*. Jednak również tworzy metodę *odległość*, której zadaniem jest obliczenie odległości danego punktu od początku układu współrzędnych. Metoda działa analogicznie do metody z listingu 3.11, operuje jednak na polach obiektu typu Punkt, a więc na właściwościach *this.x* i *this.y*. Każdy nowo utworzony obiekt typu Punkt będzie zatem zawierał zarówno pola *x* i *y*, jak i metodę *odległość*.

O tym, że tak jest, przekonujemy się, tworząc dwa obiekty, *punkt1* i *punkt2*, a następnie używając metody *odległość* każdego z nich do pobrania ich odległości od początku układu współrzędnych. Pobrane wartości są zapisywane w zmiennych pomocniczych *d1* i *d2*, które następnie używane są w instrukcjach wyświetlających wyniki na ekranie.

Zastosowane w powyższym przykładzie podejście jest stosunkowo proste, wygodne i pozwala na tworzenie spójnych obiektów wykonujących rozmaite czynności. Ma ono jednak pewną wadę. Zwróćmy uwagę na zdanie, które pojawiło się dwa akapity wyżej: „Każdy nowo utworzony obiekt [...] będzie [...] zawierał zarówno pola *x* i *y*, jak i metodę *odległość*”. To, że każdy obiekt będzie zawierał pola *x* i *y*, nie powinno budzić żadnych wątpliwości. Tak musi być, bowiem każdy z obiektów może mieć inne

wartości x i y . Czy jednak każdy obiekt musi posiadać swoją własną kopię metody odległość? Przecież będzie taka sama dla każdego punktu, niezależnie od jego współrzędnych.

Jeśli w powyższym przykładzie utworzymy 1000 obiektów typu Punkt, to w każdym z nich znajdzie się kopia kodu metody odległość. Będzie to więc zwyczajne marnotrawstwo zasobów systemowych. Na szczęście, można tego uniknąć, jeśli użyjemy tzw. prototypu. Musimy tu powrócić do opisu tego, co dzieje się po użyciu operatora new. Dotychczas prezentowany opis nie był bowiem w pełni kompletny.

Użycie operatora new powoduje utworzenie pustego obiektu oraz wywołanie funkcji konstruktora jako metody tego obiektu. To dzięki temu możemy w tej funkcji używać odwołań this. Oprócz tego, ustalany jest również prototyp obiektu. Wartością prototypu jest wartość właściwości o nazwie prototype funkcji będącej konstruktorem. Każda funkcja posiada taką właściwość, jest ona (właściwość) tworzona automatycznie, gdy tylko w kodzie pojawia się definicja funkcji.

Wartością początkową prototypu jest obiekt zawierający tylko jedną właściwość o nazwie constructor. Właściwość ta wskazuje funkcję będącą konstruktorem, z którą ów prototyp jest powiązany. Jednak do prototypu możemy dodawać kolejne właściwości i będą one dostępne dla każdego obiektu tworzonego za pomocą danego konstruktora.

Powyższy opis zapewne wydaje się dosyć zagmatwany. Wróćmy więc do przykładu. W kodzie z listingu 3.20 mieliśmy funkcję konstruktora o nazwie Punkt. A zatem jest ona obiektem przypisanym zmiennej globalnej Punkt lub — mówiąc prościej — mamy obiekt Punkt reprezentujący pewną funkcję. Obiekt ten zawiera właściwość prototype, która początkowo ma tylko jedną swoją właściwość. Jest nią constructor i wskazuje funkcję Punkt, czyli obiekt Punkt. Można powiedzieć, że te wskazania są zapętlone. Zobrazowano to na rysunku 3.12. Aby się przekonać, że tak jest w istocie, można do kodu dodać instrukcję:

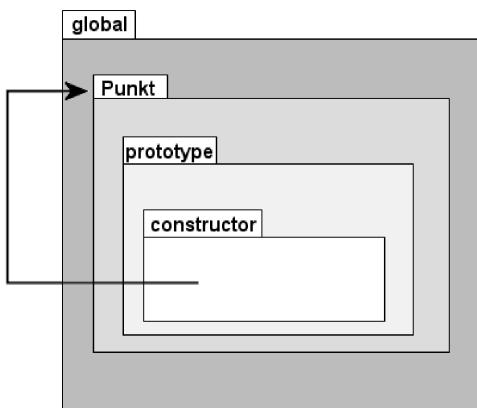
```
alert(Punkt.prototype.constructor);
```

Na ekranie zobaczymy wtedy treść funkcji Punkt.

Rysunek 3.12.

Właściwość

constructor wskazuje
obiekt funkcji



Kiedy do prototypu dodamy nową funkcję albo właściwość, będzie ona dostępna dla wszystkich tworzonych za pomocą danego konstruktora obiektów. Nie będą one jednak kopiowane do tych obiektów. A więc w ten sposób można usunąć niedogodność opisaną przy omawianiu kodu z listingu 3.20. Jak zatem powinna wyglądać definicja metody odległość? Można ją zobaczyć na listingu 3.21.

Listing 3.21. Użycie prototypu

```
function Punkt(x, y)
{
    this.x = x;
    this.y = y;
}

Punkt.prototype.odległość = function()
{
    return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

Jeśli definicję funkcji Punkt z listingu 3.20 wymienimy na tę z listingu 3.21, skrypt będzie działał tak samo, ale każdy utworzony obiekt typu Punkt będzie korzystał z funkcji odległość zdefiniowanej w prototypie. Tak więc prototyp do doskonałe miejsce do definiowania metod wspólnych dla wszystkich obiektów danego typu. Można w nim również definiować wspólne pola, choć dotyczyć to będzie raczej jedynie stałych specyficznych dla danych obiektów.

Musimy też tu wyraźnie rozgraniczyć to, co jest dostępne w prototypie i wspólne dla całej klasy obiektów, od tego, co jest specyficzne dla każdego pojedynczego obiektu. W przykładzie z obiektami typu Punkt pola x i y były zdefiniowane w konstruktorze, a zatem każdy obiekt typu Punkt operował na swoich własnych polach. To nie dziwi, bowiem każdy punkt może mieć inne współrzędne. Z kolei, metoda odległość znalazła się w prototypie, ponieważ obliczanie odległości jest wspólne dla wszystkich punktów.

Co się jednak stanie, jeżeli zarówno w prototypie, jak i w obiekcie zostanie zdefiniowana taka sama właściwość? Aby odpowiedzieć na to pytanie, musimy dowiedzieć się, jakie zasady obowiązują przy odczytzie i zapisie właściwości. Jeżeli chcemy odczytać daną właściwość, interpreter najpierw sprawdza, czy dany obiekt ją zawiera. Jeżeli tak, pobiera jej wartość. Jeżeli nie, sprawdza, czy prototyp danego obiektu zawiera tę właściwość. Jeżeli zawiera, jest pobierana, jeśli nie, zwracana jest wartość undefined.

Przy zapisie jest nieco inaczej. Najpierw badane jest, czy obiekt zawiera daną właściwość. Jeżeli tak, jej wartość jest zmieniana. Jeżeli nie, właściwość jest tworzona. Oznacza to, że nigdy nie zmodyfikujemy prototypu, zapisując właściwość obiektu. Dzięki temu jeden obiekt nie może zmodyfikować właściwości specyficznych dla wszystkich obiektów.

Zauważmy też, że z tych opisów wynika, iż właściwość zawarta w obiekcie zawsze przesyłana tę, która znajduje się w prototypie. Zobaczmy, jak to będzie wyglądało na konkretnym przykładzie, który zamieszczono na listingu 3.22. Po jego uruchomieniu ujrzymy widok przedstawiony na rysunku 3.13. Jak działa ten skrypt?

Listing 3.22. Przesłanianie właściwości

```

var str = "";
function Obiekt(x, y)
{
    this.x = x;
    this.y = y;
}
Obiekt.prototype.x = 10;
Obiekt.prototype.z = 30;

function getXYZStr(obj)
{
    var str = "x = " + obj.x;
    str += ", y = " + obj.y;
    str += ", z = " + obj.z;
    return str;
}

var obiekt1 = new Obiekt(1, 2);
var obiekt2 = new Obiekt(3, 4);

str += "Po utworzeniu obiektów: <br />";
str += "obiekt1: " + getXYZStr(obiekt1) + "<br />";
str += "obiekt2: " + getXYZStr(obiekt2) + "<br />";

obiekt1.x = 20;
obiekt1.z = 40;
obiekt2.y = 5;

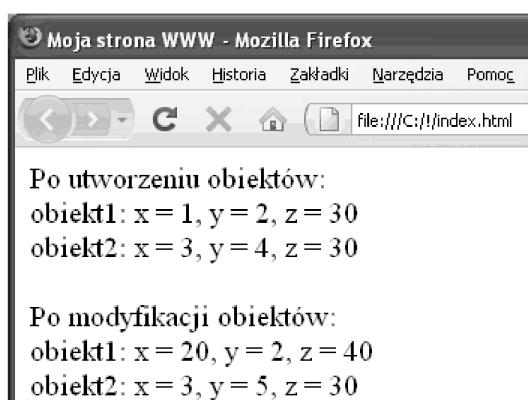
str += "<br />Po modyfikacji obiektów: <br />";
str += "obiekt1: " + getXYZStr(obiekt1) + "<br />";
str += "obiekt2: " + getXYZStr(obiekt2) + "<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Rysunek 3.13.

*Ilustracja
przesłaniania
właściwości obiektów*



Konstruktor obiektów typu `Obiekt` tworzy dwa pola o nazwach `x` i `y`. Przypisuje im też wartości swoich argumentów o takich samych nazwach. Każdy obiekt tego typu będzie posiadał swoje własne i niezależne od innych pola `x` i `y`. Prototypowi obiektów typu `Obiekt` również zostały przypisane dwie właściwości, są to `x` o wartości 10 i `z` o wartości 30:

```
Obiekt.prototype.x = 10;  
Obiekt.prototype.z = 30;
```

Zadaniem pomocniczej funkcji `getXYZStr` jest pobranie wartości właściwości `x`, `y`, z obiektu typu `Obiekt` przekazanego jej w postaci argumentu i zwrócenie ciągu znaków zawierającego komunikat określający te wartości. Wykonuje więc podobne zadanie jak funkcja `getXYStr` z listingu 3.19 (choć korzysta z nieco innej techniki).

W dalszej części kodu zostały utworzone dwa obiekty typu `Obiekt`:

```
var obiekt1 = new Obiekt(1, 2);  
var obiekt2 = new Obiekt(3, 4);
```

Wartości ich właściwości zostały wyświetcone na ekranie (dopisane do zmiennej `str`). W pierwszym przypadku są to 1, 2, 30, a w drugim — 3, 4, 30. Widzimy więc wyraźnie, że pola `x` i `y` są niezależne dla każdego obiektu, każdy obiekt posiada bowiem swoje własne właściwości `x` i `y`. Ponieważ zostały one zdefiniowane w konstruktorze, zawarte w prototype pole `x` nie ma żadnego wpływu na obiekty. Inaczej jest z polem `z`. Ponieważ nie powstało w konstruktorze, przy odczytcie danych jego wartość jest pobierana z prototypu i jest wspólna dla wszystkich obiektów typu `Obiekt`.

Po wyświetleniu danych została przeprowadzona modyfikacja właściwości obiektów:

```
obiekt1.x = 20;  
obiekt1.z = 40;  
obiekt2.y = 5;
```

A dane zostały ponownie wyświetcone. Tym razem wynikiem dla obiektu `obiekt1` są wartości 20, 2, 40, a dla obiektu `obiekt2` — 5, 4, 30. Dlaczego? Otóż, instrukcja przypisująca wartość 20 polu `x` obiektu `obiekt1` działa wyłącznie na pole zawarte w tym obiekcie. Podobnie jest z instrukcją przypisującą polu `z` wartość 40. Co prawda, takie pole nie istnieje w obiekcie `obiekt1` (a jedynie w prototype), ale zostanie ono w tej instrukcji utworzone. Pamiętamy przecież, że przypisanie wartości właściwości obiektu nie wpływa na pola prototypu, a więc i na stan innych obiektów. Trzecia instrukcja to już zwyczajna zmiana wartości pola `y` obiektu `obiekt2` na 5. Oczywiście, wpływa ona wyłącznie na stan tego obiektu.

Ostatecznie, po wykonaniu tych przypisań sytuacja będzie następująca.

- ◆ W prototype będą istniały dwa pola, `x` i `y`; pierwsze o wartości 10, drugie — 30. Wartości te nie mogą być zmienione w jakikolwiek sposób przez obiekty typu `Obiekt`. Co więcej, pole `x` nie może też być odczytane, gdyż jest trwale przesłonięte przez definicję pola `x` w konstruktorze.

- ◆ W obiekcie obiekt1 będą istniały trzy pola pochodzące z tego obiektu, czyli `x`, `y` i `z`. Pola `x` i `z` będą przesyłały pola o tych samych nazwach z prototypu. Tym samym poprzez obiekt obiekt1 nie będzie można odczytać żadnych pól pochodzących z prototypu.
- ◆ W obiekcie obiekt2 będą istniały dwa pola pochodzące z tego obiektu, `x` i `y`, oraz pole `z` pochodzące z prototypu.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 9.1.

Zmień kod z listingu 3.16 tak, aby funkcje `suma` i `różnica` nie były przypisywane zmiennej, ale definiowane w sposób klasyczny.

Ćwiczenie 9.2.

Napisz funkcję, która przyjmuje dowolną liczbę argumentów liczbowych i zwraca wartość największego z nich.

Ćwiczenie 9.3.

W kodzie z listingu 3.19 zmień konstruktor obiektu `Punkt`, tak aby samodzielnie tworzył nowy obiekt i zwracał go jako rezultat swojego działania.

Ćwiczenie 9.4.

Napisz kod konstruktora obiektu reprezentującego okrąg na płaszczyźnie. Uwzględnij metody zwracające pole i obwód okręgu. Metody te umieść w konstruktorze.

Ćwiczenie 9.5.

Napisz kod konstruktora obiektu reprezentującego prostokąt na płaszczyźnie. Uwzględnij metody zwracające pole i obwód prostokąta. Metody te umieść w prototypie.

Lekcja 10. Tablice

Tablice to struktury danych pozwalające na przechowywanie uporządkowanego zbioru elementów. Najprostszą tablicę można sobie wyobrazić jako wektor elementów, taki jak zaprezentowany na rysunku 3.14. Poszczególne pola danych nazywamy komórkami, a każda komórka ma identyfikujący ją indeks (mówimy wtedy o tablicach indeksowanych numerycznie). Pierwsza komórka (zawierająca *wartość 1*) ma indeks 0, druga — indeks 1, trzecia — indeks 2 itd.

Rysunek 3.14.

Struktura
prostej tablicy

wartość	wartość	wartość	wartość	wartość
1	2	3	4	5

W tej lekcji poznamy sposoby tworzenia tablic, zapisywania, odczytywania i zmianiania zawartych w nich danych, a także predefiniowane operacje, która można wykonywać na tych strukturach.

Jak tworzymy tablice?

Tablice w języku JavaScript są obiektami. Można je tworzyć na dwa sposoby: klasycznie, z użyciem nawiasu kwadratowego, oraz przez formalne wywołanie konstruktora (czyli zastosowanie słowa Array). W pierwszym przypadku schematyczna konstrukcja jest następująca:

```
var nazwa_tablicy = [element1, element2, ..., elementN];
```

W ten sposób powstaje N-elementowa tablica, w której w kolejnych komórkach zostały zapisane poszczególne elementy wymienione w nawiasie kwadratowym. Jeśli chcemy zwiększyć czytelność instrukcji, możemy ją rozbić na większą liczbę wierszy:

```
var nazwa_tablicy = [  
    element1,  
    element2,  
    ...  
    elementN  
];
```

Wartość każdej komórki może być dowolnego typu. Aby np. utworzyć 5-elementową tablicę, w której w kolejnych komórkach zostały zapisane następujące po sobie liczby całkowite, można użyć instrukcji:

```
var tablica = [1, 2, 3, 4, 5];
```

Gdybyśmy natomiast chcieli umieścić w niej trzy ciągi znaków, można zastosować konstrukcję:

```
var tablica = ["abc", "def", "ghj"];
```

W jednej tablicy mogą się też znajdować wartości różnych typów, np. wartość całkowita, rzeczywista i ciąg znaków:

```
var tablica = [18, 34.12, "hello"];
```

Istnieje również możliwość utworzenia pustej tablicy, do której w dalszej części skryptu będą zapisywane dane. Wystarczy użyć instrukcji:

```
var tablica = [];
```

Składnia z nawiasem kwadratowym dopuszcza pominięcie w definicji niektórych elementów. Powstają wtedy komórki o wartości undefined. Pusty element zostanie utworzony, jeśli pomiędzy dwoma przecinkami nie umieścimy żadnej wartości. I tak instrukcja:

```
var tablica = [1.,3.,5];
```

spowoduje utworzenie tablicy o strukturze zaprezentowanej na rysunku 3.15.

Rysunek 3.15.

Tablica z elementami niezdefiniowanymi

1	undefined	3	undefined	5
---	-----------	---	-----------	---

Drugi sposób konstruowania to formalne wywołanie konstruktora typu tablicowego, schematycznie:

```
var tablica = new Array(element1, element2, ..., elementN);
```

Oto przykład:

```
var tablica = new Array(1, 2, 3, 4, 5);
```

Podobnie jak w składni z nawiasem kwadratowym, przy dużej liczbie elementów dla zwiększenia czytelności taką instrukcję można zapisać w kilku wierszach:

```
var tablica = new Array(
    element1,
    element2,
    ...
    elementN
);
```

Można utworzyć pustą tablicę:

```
var tablica = new Array();
```

Szczególną uwagę należy natomiast zwrócić na konstrukcję w postaci:

```
var tablica = new Array(wartość_całkowita);
```

Jej znaczenie jest bowiem odmienne, niż mogłoby się wydawać. Nie oznacza ona bowiem utworzenia tablicy 1-elementowej i zapisania w jej komórce wartości *wartość_całkowita*, ale utworzenie tablicy o początkowej liczbie komórek równej parametrowi *wartość_całkowita*. Znaczy to, że jeśli zastosujemy instrukcję:

```
var tablica = new Array(10);
```

powstanie 10-elementowa tablica, natomiast każda jej komórka będzie początkowo zawierała wartość undefined.

Jak zapisywać i odczytywać dane?

Odczyt danych z tablicy jest bardzo prosty. Wystarczy podać indeks żądanego elementu w nawiasie kwadratowym, schematycznie:

```
var zmienna = tablica[indeks];
```

Oczywiście, odczytywana zawartość nie musi być przypisywana zmiennej, ale może być bezpośrednio użyta w instrukcji alert czy innym wyrażeniu, np.:

```
alert(tablica[indeks]);
if(tablica[indeks] == wartość){/*tutaj dalsze instrukcje*/};
```

Jeżeli więc chcemy odczytać zawartość komórki o indeksie 3 z tablicy o nazwie liczby i przypisać odczytane dane zmiennej value, powinniśmy skorzystać z instrukcji:

```
var value = liczby[3];
```

Należy przy tym zwrócić szczególną uwagę na sposób indeksowania tablicy. Indeksowanie rozpoczyna się od 0, a nie od 1. To oznacza, że pierwszy element (pierwsza komórka) tablicy ma indeks 0. Wynika z tego, że tablice utworzone za pomocą instrukcji:

```
var liczby = [1, 2, 3, 4, 5];
var znaki = ["abc", "def", "ghj"];
```

będą indeksowane w sposób przedstawiony na rysunku 3.16.

Rysunek 3.16.
Indeksowanie tablicy
zaczyna się od 0

tablica liczby

1	2	3	4	5
indeks: 0	1	2	3	4

tablica znaki

abc	def	ghj
indeks: 0	1	2

Utwórzmy zatem skrypt, w którym powstanie tablica zawierająca nazwy, np. kolorów, odczytajmy zawartość tej tablicy i wyświetlmy te dane w oknie przeglądarki. Tak działający skrypt przyjmie postać widoczną na listingu 3.23.

Listing 3.23. Tworzenie i odczytywanie tablicy

```
var kolory = ["czerwony", "zielony", "niebieski"];
var str = "W tablicy kolory znajdują się dane:";

str += "<br />kolory[0] = ";
str += kolory[0];
```

```

str += "<br />kolory[1] = ";
str += kolory[1];
str += "<br />kolory[2] = ";
str += kolory[2];
str += "<br />";

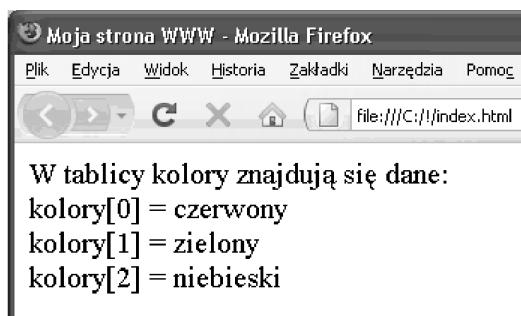
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Została tu utworzona 3-elementowa tablica, której kolejnym komórkom zostały przypisane ciągi znaków określające kolory: czerwony, zielony i niebieski. Tablica (obiekt tablicy) została przypisana zmiennej kolory. Upraszczając terminologię, można powiedzieć, że powstała tablica kolory. Aby uzyskać dostęp do wartości zapisanej w danej komórce, należy podać jej numer (indeks) w nawiasie kwadratowym występującym za nazwą tablicy. Trzeba przy tym pamiętać, że indeksowanie zaczyna się od 0, co oznacza, że indeksem pierwszej komórki jest 0, a NIE 1. A zatem, aby odczytać zawartość pierwszej komórki, napisaliśmy kolory[0], drugiej — kolory[1], a trzeciej — kolory[2]. Dzięki temu po uruchomieniu skryptu na ekranie zobaczymy widok przedstawiony na rysunku 3.17.

Rysunek 3.17.

Zawartość tablicy
kolory wyświetlona
w przeglądarce



W celu utworzenia nowej komórki tablicy należy po prostu podać jej indeks i przypisać wartość, schematycznie:

```
tablica[indeks] = wartość;
```

Może to być wartość dowolnego typu. A zatem, jeśli w tablicy numery chcemy dodać czwarty element o wartości 52, użyjemy instrukcji:

```
numery[3] = 52;
```

Oczywiście, użycie indeksu 3 nie jest błędem, skoro bowiem indeksowanie rozpoczyna się od 0, to czwarty element ma indeks 3.

W identyczny sposób można zmienić istniejącą zawartość danej komórki. Jeśli użyjemy powyższej instrukcji, a element o indeksie 3 istnieje, jego zawartość zostanie utracona, a na to miejsce wprowadzona wartość 52. Ilustruje to przykład widoczny na listingu 3.24.

Listing 3.24. Zapis elementów tablicy

```
var tab = new Array();
var str = "";

tab[0] = "czarny";
tab[1] = "zielony";
tab[2] = "niebieski";

str = "Pierwotna zawartość tablicy tab:";

str += "<br />tab[0] = "
str += tab[0];
str += "<br />tab[1] = ";
str += tab[1];
str += "<br />tab[2] = ";
str += tab[2];
str += "<br />";

tab[0] = 111;
tab[1] = 222;
tab[2] = 333;

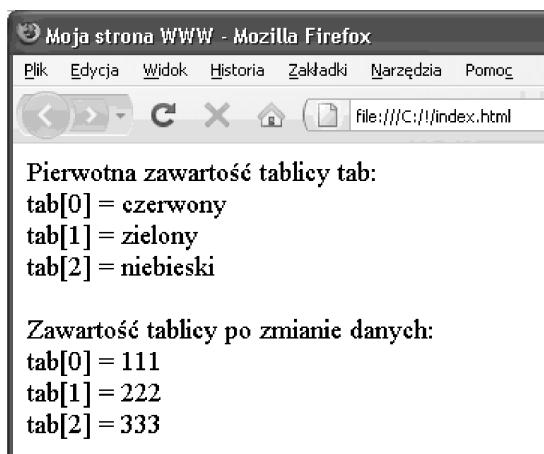
str += "<br />Zawartość tablicy po zmianie danych:";

str += "<br />tab[0] = "
str += tab[0];
str += "<br />tab[1] = ";
str += tab[1];
str += "<br />tab[2] = ";
str += tab[2];
str += "<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Na początku, przy użyciu składni ze słowem `Array`, powstała pusta tablica. Została ona przypisana zmiennej `tab`. Trzem kolejnym komórkom tej tablicy przypisano trzy ciągi znaków określające różne kolory. Następnie cała zawartość została odczytana i przypisana zmiennej `str` (tym samym pojawi się na ekranie). W drugiej części kodu komórkom o indeksach 0, 1 i 2 zostały przypisane wartości liczbowe — tym samym usunięte zostały znajdujące się w nich wcześniej ciągi znaków. Nowa zawartość tablicy również została odczytana i przypisana zmiennej `str`. A zatem po uruchomieniu skryptu zobaczymy widok zaprezentowany na rysunku 3.18. Widzimy więc, że zawartość komórek może być dowolnie zmieniana, a przypisywane im dane mogą mieć różne typy.

Rysunek 3.18.
Zawartość tablicy
została wymieniona



Indeksy i właściwości tablic

Indeks tablicy musi być wartością zawierającą się w przedziale od 0 do $2^{32} - 1$. Indeksy nie muszą występować jeden po drugim, mogą między nimi występować „dziury”, tzn. można wykonać przykładowe instrukcje:

```
var tab = new Array();
tab[0] = 100;
tab[20] = 200;
```

Powstanie wtedy tablica o dwóch komórkach. Pierwsza będzie miała indeks 0, a druga 21. Pozostałych komórek po prostu nie będzie (oczywiście, można je dodać w dalszej części kodu), choć z bardziej praktycznego punktu widzenia powinniśmy powiedzieć, że występujące między 0 a 20. indeksem komórki mają wartość `undefined`. O co chodzi? Typowa tablica w klasycznym języku programowania musi zachować ciągłość indeksacji. Jeśli nawet część komórek miałaby być pusta, muszą one formalnie wystąpić. Taka jest natura samej tablicy. Ułatwia to automatyczne przetwarzanie danych, wiadomo bowiem, że pomiędzy indeksami x a y na pewno występuje $y - x - 1$ elementów. W językach skryptowych zazwyczaj nie ma takich sztywnych ograniczeń i z reguły tablicami nazywamy obiekty przechowujące zbiory uporządkowanych elementów (czy też par indeks-wartość). Nie ma potrzeby deklarowania z góry rozmiaru tablicy, elementy można dodawać dynamicznie, w dowolnym jej miejscu. Problem pojawia się wtedy, gdy chcemy przetwarzać taką tablicę. Jak bowiem dowiedzieć się, które elementy istnieją, a które nie? Zastosowane w JavaScriptie rozwiązanie jest takie, że odwołanie się do nieistniejącego elementu powoduje zwrócenie wartości `undefined`.

Przydatną właściwością tablicy jest `length`. Określa ona jej długość, czyli liczbę komórek. Jednak w świetle tego, co zostało napisane wyżej, musimy pamiętać, że długość jest prawidłowa tylko dla tablic o ciągłej indeksacji (innych w książce nie będziemy wykorzystywać). A więc jeśli mamy tablicę w postaci:

```
var liczby = [1, 2, 3, 4, 5];
```

to odwołanie `tab.length` da w wyniku wartość 5 — tablica ma bowiem pięć elementów. Jaką jednak wartość będzie miała ta właściwość dla tablic z przerwami w indeksacji? Otóż, o jeden większą niż największy indeks. Zatem po wykonaniu instrukcji:

```
tab = [1];
tab[30] = 100;
```

właściwość `length` tablicy `tab` będzie miała wartość 31. To drugi powód, dla którego w praktyce nieistniejące formalnie elementy między indeksami można traktować jak elementy o wartości `undefined`.

Ponieważ tablice są obiektami, można im przypisywać właściwości, i to używając składni z nawiasem kwadratowym. A to może prowadzić do nieporozumień. Oto przykład. Na początku lekcji została podana informacja, że indeks tablicy musi się zawsze zatrzymać w przedziale od 0 do $2^{32} - 1$. Tymczasem można użyć takiej konstrukcji:

```
var tab = [];
tab[-1] = 10;
```

i zostanie ona wykonana. Jednak w ten sposób nie powstała komórka tablicy o indeksie -1. Równie dobrze można bowiem użyć konstrukcji:

```
tab[3.14] = 10;
```

czy:

```
tab["abcd"] = 10;
```

W ten sposób nie definiujemy komórkę tablicy `tab`, ale właściwość obiektu `tab`. Drukując mówiąc, jeśli podana wartość przekracza dopuszczalny zakres lub nie jest liczbą całkowitą, zostanie zamieniona na ciąg znaków i potraktowana jako właściwość obiektu tablicy. Nie wpłynie to więc np. na właściwość `length`. Rozważmy przykład:

```
var tab = [1, 2, 3];
tab[-5] = 10;
tab[3.14] = 20;
var rozmiar = tab.length;
```

Jaka będzie wartość zmiennej `rozmiar`? Zgodnie z powyższym będzie to 3, bowiem została zdefiniowana 3-elementowa tablica `tab`, a następnie obiektovi tej tablicy zostały dodatkowo przypisane właściwości o nazwach "-5" i "3.14".

Użycie pętli

Podane wyżej sposoby odczytu i zapisu tablic mogłyby wystarczyć tylko wtedy, gdyby ilość przechowywanych danych zawsze była niewielka. Jeśli danych jest dużo lub też ich ilość nie jest z góry znana (a z takimi sytuacjami bardzo często się spotyka), niezbędne będzie wprowadzenie zautomatyzowanego przetwarzania tablic. W tym celu użyjemy pętli. Posłużą one zarówno do zapisu, jak i odczytu. Założymy, że chcemy zapisać w tablicy 100 kolejnych liczb całkowitych, a następnie wyświetlić je na ekranie. Nic prostszego — wystarczy użyć pętli `for`. Spójrzmy na skrypt widoczny na listingu 3.25. Realizuje takie właśnie zadanie.

Listing 3.25. Użycie pętli for do obsługi tablicy

```

var str = "";

var tab = [];
for(i = 0; i < 100; i++){
    tab[i] = i;
}

for(i = 0; i < 100; i++){
    str += tab[i];
    str += " ";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Na początku została utworzona pusta tablica tab. Następnie w pętli for wypełniono ją danymi. Zmienna iteracyjna i tej pętli zmienia się od 0 do 99 i została użyta zarówno jako indeks kolejnej komórki tablicy, jak i wartość tej komórki, tzn. komórka o indeksie 0 zawiera wartość 0, komórka o indeksie 1 — wartość 1 itd.

Druga pętla for odpowiada za odczytanie całej tablicy. Jej zmienna iteracyjna i również zmienia się od 0 do 99, co pozwala odczytać zawartość wszystkich komórek. Odczytane wartości są dopisywane do zmiennej str:

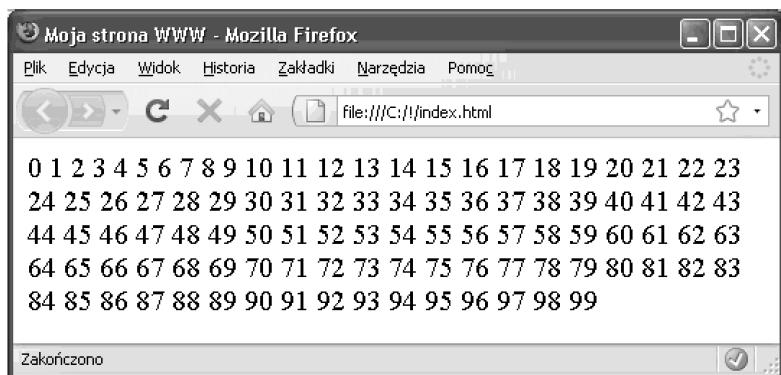
```
str += tab[i];
```

A po każdej z nich dopisywana jest również spacja:

```
str += " ";
```

Dzięki temu po uruchomieniu skryptu zobaczymy ciąg liczb przedstawiony na rysunku 3.19.

Rysunek 3.19.
Kolejnych 100 liczb
pobranych z tablicy



W powyższym przykładzie odczyt danych odbywał się w pętli for, w której końcowa wartość zmiennej iteracyjnej była z góry znana i wynosiła 99. Było to możliwe, bo wiemy znałyśmy liczbę elementów tablicy (100). Co zrobić, jeśli wartość ta nie jest z góry znana? Odpowiedź została już częściowo zawarta w poprzedniej części lekcji.

Skoro bowiem każda tablica zawiera właściwość `length` określającą liczbę jej elementów, można tę właściwość wykorzystać w pętli `for` do skonstruowania warunku zakończenia. Druga pętla `for` z listingu 3.25 mogłaby zatem mieć postać:

```
for(i = 0; i < tab.length; i++){
    str += tab[i];
    str += " ";
}
```

Takie rozwiązanie jest dużo bardziej elastyczne (w ten sposób odczytamy całą zawartość tablicy, niezależnie od liczby przechowywanych przez nią elementów), a efekt będzie taki sam jak we wcześniejszym przykładzie.

Korzystając z takich rozwiązań, trzeba jednak wziąć pod uwagę tablice o nieliniowej indeksacji (czyli takie, które mają przerwy między kolejnymi indeksami, np. 1, 5, 8). Spójrzmy na listing 3.26.

Listing 3.26. Błędny odczyt tablicy o nieliniowej indeksacji

```
var str = "";

var tab = [];
for(i = 0; i < 100; i += 2){
    tab[i] = i;
}

for(i = 0; i < tab.length; i++){
    str += tab[i];
    str += " ";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Jest podobny do przykładu z listingu 3.25. Tu również powstaje tablica, której zawartość jest odczytywana i wyświetlana na ekranie. Po uruchomieniu kodu zobaczymy jednak obraz przedstawiony na rysunku 3.20. Zapewne nie o to nam chodziło. Oprócz zawartości tablicy, pojawiły się również słowa `undefined`. Skąd taki efekt? Przyjrzyjmy się pętli tworzącej tablicę, a szczegółowo wyrażeniu modyfikującemu. Ma ono postać:

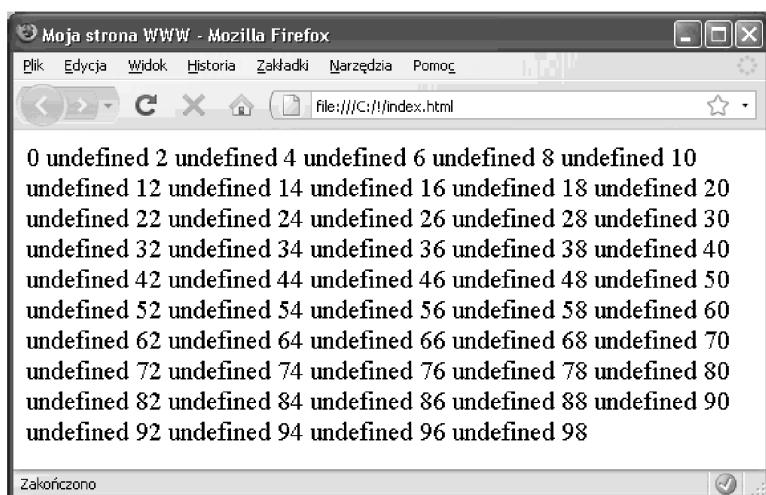
```
i += 2
```

Oznacza to, że w każdym przebiegu pętli wartość zmiennej iteracyjnej `i` jest zwiększana o 2. Dlatego też taka tablica ma dwa razy mniej komórek niż poprzednia. Kolejne indeksy (oraz wartości) to: 0, 2, 4, 6 itd.

Z kolei pętla odczytująca przebiega od 0 aż do wartości wskazywanej przez `tab.length - 1` (warunek `i < tab.length`). Jak wiemy, wartość właściwości `length` jest o jeden większa niż maksymalny indeks tablicy (indeks ostatniej komórki). Indeksem ostatniej komórki jest 98, więc `length` ma wartość 99. To oznacza, że próbujemy odczytać wszystkie wartości, począwszy od zawartej pod indeksem 0, a skończywszy na

zawartej pod indeksem 98. Niestety, indeksów 1, 3, 5 itd. nie ma (inaczej: mają wartość undefined). Próba ich odczytu kończy się więc zwróceniem wartości undefined. Dlatego też na ekranie pojawił się widok przedstawiony na rysunku 3.20.

Rysunek 3.20.
Nieprawidłowe
pobranie danych
z tablicy o nielinowej
indeksacji



Jak tego uniknąć? W pętli typu for trzeba badać stan odczytywanej komórki. Gdy stwierdzimy, że jest to undefined, trzeba pominąć instrukcje wyświetlające dane. Można to zrobić np. w następujący sposób:

```
for(i = 0; i < tab.length; i++){
    if(tab[i] != undefined){
        str += tab[i];
        str += " ";
    }
}
```

Za pomocą instrukcji if badamy tutaj warunek `tab[i] != undefined`. Będzie on prawdziwy, gdy dana komórka będzie różna od undefined. Wtedy też zostaną wykonane instrukcje dodające dane do zmiennej str. Jeśli zaś warunek będzie fałszywy, instrukcje te będą pominięte i zostanie wykonana kolejna iteracja pętli.

Zamiast stosować pętlę for i instrukcję if, można też użyć pętli typu for...in. Pozwoli ona odczytać wszystkie indeksy tablicy:

```
for(var indeks in tab){
    str += tab[indeks];
    str += " ";
}
```

W takim przypadku nie musimy się też przejmować rozmiarem tablicy — wszystkie komórki zostaną automatycznie uwzględnione. Musimy jednak uważać. Jeżeli bowiem w obiekcie tablicy zadeklarowaliśmy jakieś właściwości (co nie jest dobrą praktyką), zostaną one uwzględnione przez pętlę for...in. Dla przykładu przeanalizujmy kod widoczny na listingu 3.27.

Listing 3.27. Korzystanie z pętli *for...in*

```
var str = "";

var tab = [];
tab['abc'] = 123;

for(i = 0; i < 5; i++){
    tab[i] = i;
}

tab[3.14] = 901;

for(var indeks in tab){
    str += tab[indeks];
    str += " ";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Najpierw został utworzony obiekt tablicy tab. Następnie za pomocą instrukcji:

```
tab['abc'] = 123;
```

powstała właściwość tego obiektu. Nadaliśmy jej nazwę abc i wartość 123. Dalej w pętli for powstało 5 komórek o indeksach od 0 do 4. Za pętlą znalazła się instrukcja:

```
tab[3.14] = 901;
```

Spowodowała ona utworzenie kolejnej właściwości o nazwie "3.14" (wartość rzeczywista 3.14 została skonwertowana niejawnie na ciąg znaków "3.14") i przypisanie jej wartości 901. Całość możemy potraktować tak, jakby powstał obiekt tab z właściwościami abc, 0, 1, 2, 3, 4, 3.14. Pętla for...in obejmie wszystkie te właściwości (zdefiniowane przez nas w kodzie skryptu), a nie tylko indeksy od 0 do 4. Zatem na ekranie ukąże się ciąg 123 0 1 2 3 4 901.

Operacje na tablicach

Tablice są obiektami typu Array. W związku z tym, zawierają zestaw predefiniowanych metod pozwalających na wykonywanie różnorodnych operacji. Metody zostały zebrane w tabeli 3.5. W tej części lekcji zobaczymy, jak używać niektórych z nich.

Łączenie tablic

Dwie tablice można ze sobą połączyć, korzystając z metody concat. Jej schematyczne wywołanie ma postać:

```
tablica2 = tablica1.concat(lista elementów);
```

Tabela 3.5. Metody obiektu Array

Metoda	Wymyłanie	Opis	Dostępność
concat	<code>tablica1.concat →(tablica2)</code>	Wykonuje łączenie dwóch tablic. Tablice <code>tablica1</code> i <code>tablica2</code> zostaną dodane do siebie, a powstały w ten sposób obiekt zostanie zwrócony w wyniku działania funkcji.	od JavaScript 1.2 i JScript 3.0
join	<code>tablica.join(separator)</code>	Zwraca zawartość wszystkich komórek tablicy w postaci ciągu znaków, w którym poszczególne wartości oddzielone są znakami separatora.	od JavaScript 1.1 i JScript 3.0
pop	<code>tablica.pop()</code>	Pobiera z tablicy ostatni element i jednocześnie usuwa go.	od JavaScript 1.2 i JScript 5.0
push	<code>tablica.push(element)</code>	Dodaje na końcu tablicy nowy element.	od JavaScript 1.2 i JScript 5.5
reverse	<code>tablica.reverse()</code>	Odwraca kolejność komórek tablicy.	od JavaScript 1.1 i JScript 3.0
shift	<code>tablica.shift()</code>	Pobiera i usuwa pierwszy element tablicy.	od JavaScript 1.2 i JScript 5.5
slice	<code>tablica.slice →(start, [end])</code>	Zwraca nową tablicę zawierającą elementy tablicy <code>tablica</code> od indeksu <code>start</code> do indeksu <code>end</code> . Jeżeli parametr <code>end</code> zostanie pominięty, metoda <code>slice</code> zachowią się tak, jakby miał on wartość <code>tablica.length</code> .	od JavaScript 1.2 i JScript 3.0
sort	<code>tablica.sort →([nazwa_funkcji])</code>	Sortuje elementy tablicy. Jako argument może zostać dostarczona nazwa funkcji porównującej dwa elementy.	od JavaScript 1.1 i JScript 3.0
splice	<code>tablica.splice →(indeks, ile, →[wartość1, wartość2, →..., wartośćN])</code>	Usuwa z tablicy, począwszy od indeksu <code>indeks</code> , liczbę elementów wskazywaną przez argument <code>ile</code> . W miejsce usuniętych elementów mogą zostać wstawione nowe, wskazywane przez argumenty <code>wartość1</code> , <code>wartość2</code> itd.	od JavaScript 1.2 i JScript 5.5
toString	<code>tablica.toString()</code>	Zwraca串 znaków przedstawiający zawartość wszystkich komórek tablicy. Poszczególne wartości oddzielone są znakami przecinka. Działanie tej metody jest identyczne z wywołaniem <code>tablica.join(",")</code> .	od JavaScript 1.1 i JScript 3.0
unshift	<code>tablica.unshift →(element)</code>	Wstawia element na początku tablicy. Odwrotnością metody <code>shift</code> .	od JavaScript 1.2 i JScript 5.5

W wyniku powstanie tablica `tablica2` składająca się z elementów zawartych w tablicy `tablica1`, za którymi zostaną umieszczone wszystkie elementy z listy elementów występującej jako argument metody `concat`. Na tej liście mogą się znaleźć zarówno pojedyncze wartości, jak i inne tablice. Rozważmy kilka przykładów, zakładając, że mamy zdefiniowane w kodzie 3 tablice:

```
var tab1 = [1, 2, 3];
var tab2 = [3, 4, 5];
var tab3 = [6, 7, 8];
```

Jeżeli wykonamy operację:

```
var tab4 = tab1.concat(tab2);
```

w wyniku otrzymamy tablicę tab4 o zawartości 1, 2, 3, 3, 4, 5.

Jeżeli wykonamy operację:

```
var tab4 = tab1.concat(tab3, tab2);
```

w wyniku otrzymamy tablicę tab4 o zawartości 1, 2, 3, 6, 7, 8, 3, 4, 5.

Jeżeli wykonamy operację:

```
var tab4 = tab1.concat(4, 5);
```

w wyniku otrzymamy tablicę tab4 o zawartości 1, 2, 3, 4, 5.

Jeżeli wykonamy operację:

```
var tab4 = tab1.concat(4, 5, tab3, 9);
```

w wyniku otrzymamy tablicę tab4 o zawartości 1, 2, 3, 4, 5, 6, 7, 8, 9.

Odwracanie kolejności elementów

Jeśli chcemy odwrócić kolejność elementów w tablicy, czyli spowodować, aby pierwszy stał się ostatnim, drugim przedostatnim itd., możemy zastosować metodę reverse. Nie przyjmuje ona żadnych argumentów i zmienia kolejność elementów w bieżącej tablicy. Powinny na to zwrócić uwagę osoby programujące w innych językach skryptowych, np. PHP, gdzie najczęściej oryginalna tablica nie jest zmieniana. Jeżeli zatem utworzymy przykładową tablicę:

```
var tab = [1, 2, 3, 4];
```

to po wykonaniu instrukcji:

```
tab.reverse();
```

będzie zawierała elementy w kolejności 4, 3, 2, 1.

Dodawanie i usuwanie elementów

Metody pop, push, shift i unshift pozwalają na dodawanie i usuwanie elementów, z początku i z końca tablicy. Metoda pop pobiera element znajdujący się na końcu tablicy (jednocześnie usuwając go) i zwraca jego wartość. Tym samym tablica zostaje skrócona o ostatni element. Podobne zadanie wykonuje metoda shift, z tą różnicą, że usuwany jest pierwszy element. Wszystkie elementy zostaną też przenumerowane, czyli indeks każdego z nich zmniejszy się o jeden.

Metoda push działa odwrotnie niż pop. Dodaje elementy przekazane w postaci listy argumentów na końcu tablicy. Schematycznie operację tę można przedstawić jako:

```
tablica.push(element1, element2, ..., elementN);
```

Metoda zwraca wartość określającą liczbę elementów w powiększonej tablicy. Podobnie do push działa unshift — dodaje określzoną liczbę elementów na początku tablicy. Tablica zostanie wtedy odpowiednio przenumerowana. Wywołanie metody unshift ma schematyczną postać:

```
tablica.unshift(element1, element2, ..., elementN);
```

Działanie wszystkich wymienionych metod ilustruje skrypt widoczny na listingu 3.28.

Listing 3.28. Dodawanie i usuwanie elementów tablicy za pomocą metod

```
var str = "";
var tab = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

str += "Pierwotna zawartość tablicy: ";
for(var i in tab) str += tab[i] + " ";

val = tab.pop();
str += "<br />Wynik pierwszej operacji pop: ";
str += val;

val = tab.pop();
str += "<br />Wynik drugiej operacji pop: ";
str += val;

str += "<br />Aktualna zawartość tablicy: ";
for(var i in tab) str += tab[i] + " ";

val = tab.shift();
str += "<br />Wynik pierwszej operacji shift: ";
str += val;

val = tab.shift();
str += "<br />Wynik drugiej operacji shift: ";
str += val + "<br />";

str += "Aktualna zawartość tablicy: ";
for(var i in tab) str += tab[i] + " ";

tab.push(1, 2);
str += "<br />Zawartość tablicy po operacji push: ";
for(var i in tab) str += tab[i] + " ";

tab.unshift(9, 10);
str += "<br />Zawartość tablicy po operacji unshift: ";
for(var i in tab) str += tab[i] + " ";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

W skrypcie tworzona jest tablica tab, która początkowo zawiera uporządkowane rosnąco wartości od 1 do 10. Wykonanie dwóch operacji tab.pop(): powoduje usunięcie dwóch ostatnich wartości, a zatem pozostają komórki od 1 do 8. Następnie są wykonywane dwie operacje tab.shift():, które usuwają dwie pierwsze komórki; tym samym w tablicy pozostają wartości od 3 do 8. Należy zwrócić uwagę, że przenumerowaniu uległy również indeksy komórek. Wartość 3 znajduje się obecnie pod indeksem 0, wartość 4 pod indeksem 1 itd. Kolejną wykonywaną operacją jest tab.push(1, 2):, która powoduje dodanie na końcu tablicy dwóch komórek, pierwszej o wartości 1 i drugiej o wartości 2. Operacja tab.unshift(9, 10): powoduje natomiast dodanie na początku tablicy dwóch komórek, pierwszej o wartości 9 i drugiej o wartości 10. Ostatecznie tablica będzie zatem zawierała ciąg wartości 9, 10, 3, 4, 5, 6, 7, 8, 1, 2, co widać na rysunku 3.21.

Rysunek 3.21.

Wynik działania skryptu z listingu 3.28

```

Moja strona WWW - Mozilla Firefox
Plik Edycja Widok Historia Zakładki Narzędzia Pomoc
File:///C:/l/index.html
Pierwotna zawartość tablicy: 1 2 3 4 5 6 7 8 9 10
Wynik pierwszej operacji pop: 10
Wynik drugiej operacji pop: 9
Aktualna zawartość tablicy: 1 2 3 4 5 6 7 8
Wynik pierwszej operacji shift: 1
Wynik drugiej operacji shift: 2
Aktualna zawartość tablicy: 3 4 5 6 7 8
Zawartość tablicy po operacji push: 3 4 5 6 7 8 1 2
Zawartość tablicy po operacji unshift: 9 10 3 4 5 6 7 8 1 2
Zakończono

```

Sortowanie

Jedną z operacji często wykonywanych na tablicach jest sortowanie, czyli ustawienie elementów w danym porządku. Zobaczmy, w jaki sposób można wykonać taką operację w JavaScirpcie. Umożliwi to metoda sort. Działa ona zarówno na wartościach liczbowych, jak i na ciągach znaków. Spójrzmy na listing 3.29. Zawiera on kod sortujący dwie różne tablice.

Listing 3.29. Standardowe sortowanie tablic

```

var str = "";
var tab1 = Array(5, 7, 3, 1, 8, 2, 0, 4, 9, 6);
var tab2 = Array('jeden', 'dwa', 'trzy', 'cztery', 'pięć');

str += "Zawartość tablic przed sortowaniem: <br />";
for(var i in tab1) str += tab1[i] + " ";
str += "<br />";
for(var i in tab2) str += tab2[i] + " ";

```

```

tab1.sort();
tab2.sort();

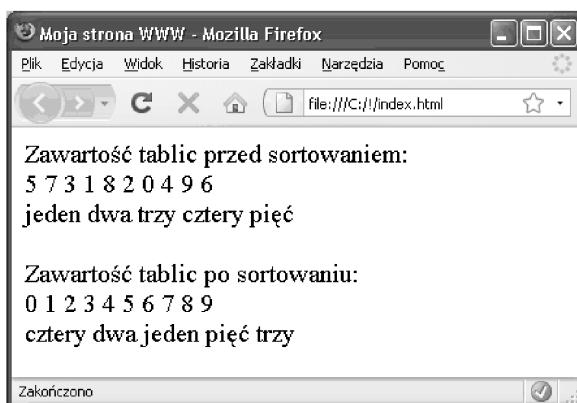
str += "<br /><br />Zawartość tablic po sortowaniu: <br />";
for(var i in tab1) str += tab1[i] + " ";
str += "<br />";
for(var i in tab2) str += tab2[i] + " ";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Efekt wykonania skryptu został przedstawiony na rysunku 3.22. Jak widać, obie tablice zostały poprawnie posortowane. Oczywiście, w tablicy tab1 mieliśmy do czynienia z sortowaniem liczb i wartości zostały ustawione od najmniejszej do największej, natomiast w tablicy tab2 nastąpiło sortowanie ciągów znaków, a zatem słowa zostały ustawione w porządku alfabetycznym.

Rysunek 3.22.
Efekt sortowania tablic



Co jednak zrobić, gdybyśmy chcieli ustawić wartości znajdujące się w pewnej tablicy w specyficznej kolejności, odmiennej od standardowego porządku? Możemy sami napisać całą funkcję wykonującą sortowanie, łatwiej jednak użyć metody sort i dodatkowej funkcji porównującej dwa elementy. Schematyczne wywołanie ma wtedy postać:

```
tablica.sort(nazwa_funkcji);
```

Taka funkcja będzie otrzymywała w postaci argumentów dwa elementy sortowanej tablicy, musi natomiast zwracać:

- ◆ wartość mniejszą od 0, jeśli pierwszy argument jest mniejszy od drugiego;
- ◆ wartość większą od 0, jeśli pierwszy argument jest większy od drugiego;
- ◆ wartość równą 0, jeśli pierwszy argument jest równy drugiemu.

Zobaczmy, jak to działa na konkretnym przykładzie. Napiszemy skrypt, który będzie sortował zawartość tablicy przechowującej liczby całkowite w taki sposób, że najpierw umieszczone będą wartości podzielne przez 2, od najmniejszej do największej, a dopiero po nich wartości niepodzielne przez 2, również od najmniejszej do największej. Zadanie to realizuje kod z listingu 3.30.

Listing 3.30. Sortowanie specjalne

```
function porównaj(e1, e2)
{
    if(e1 % 2 == 0){
        if(e2 % 2 == 0){
            return e1 - e2;
        }
        else{
            return -1;
        }
    }
    else{
        if(e2 % 2 == 0){
            return 1;
        }
        else{
            return e1 - e2;
        }
    }
}

var str = "";
var tabl = Array(5, 7, 3, 1, 8, 2, 0, 4, 9, 6);

str += "Zawartość tablicy przed sortowaniem: <br />";
for(var i in tabl) str += tabl[i] + " ";

tabl.sort(porównaj);

str += "<br /><br />Zawartość tablicy po sortowaniu: <br />";
for(var i in tabl) str += tabl[i] + " ";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

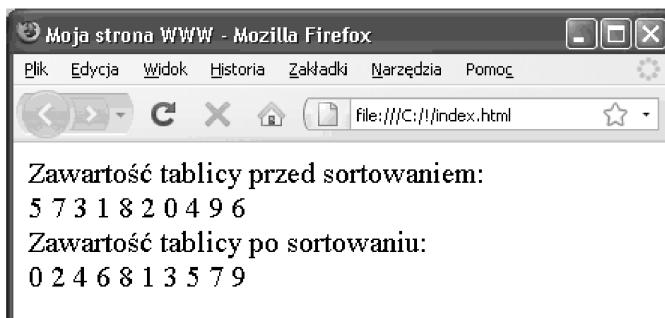
Tablica powstaje w standardowy sposób i jej początkowa zawartość jest wyświetlana na ekranie. Następnie wywoływana jest metoda `sort`, wykonująca operację sortowania, i zawartość posortowanej tablicy jest ponownie wyświetlana na ekranie. Tym samym w przeglądarce ukaże się obraz widoczny na rysunku 3.23. Jak widać, układ liczb jest zgodny z naszymi założeniami; najpierw umieszczone są liczby podzielne przez 2, od najmniejszej do największej, a za nimi liczby niepodzielne przez 2, również od najmniejszej do największej. Za takie uporządkowanie elementów tablicy odpowiada kombinacja metody `usort` i funkcji `porównaj`.

Przyjrzyjmy się zatem funkcji `porównaj`. Występuje w niej złożona, zagnieżdżona instrukcja warunkowa. Przy porównywaniu dwóch dowolnych elementów tablicy `tabl` możliwe są bowiem cztery różne sytuacje. Oto one.

1. Pierwszy argument jest podzielny przez 2 (`e1 % 2` równe 0) i drugi argument jest również podzielny przez 2 (`e2 % 2` równe 0). W takiej sytuacji należy zwrócić wartość mniejszą od 0, jeśli pierwszy argument jest mniejszy, wartość większą od 0, jeśli drugi argument jest mniejszy, lub wartość 0, jeśli argumenty są równe. Zapewnia to instrukcja `return e1 - e2;`.

Rysunek 3.23.

*Efekt
niestandardowego
sortowania*



2. Pierwszy argument jest podzielny przez 2 ($e1 \% 2$ równe 0), natomiast drugi argument nie jest podzielny przez 2 ($e2 \% 2$ różne od 0). W takiej sytuacji argument pierwszy zawsze powinien znaleźć się przed argumentem drugim, a zatem należy zwrócić wartość mniejszą od 0. Zapewnia to instrukcja `return -1;`.
3. Pierwszy argument nie jest podzielny przez 2 ($e1 \% 2$ różne od 0), a drugi argument jest podzielny przez 2 ($e2 \% 2$ równe 0). W takiej sytuacji argument pierwszy zawsze powinien znaleźć się za argumentem drugim, a zatem należy zwrócić wartość większą od 0. Zapewnia to instrukcja `return 1;`.
4. Pierwszy argument nie jest podzielny przez 2 ($e1 \% 2$ różne od 0) i drugi argument również nie jest podzielny przez 2 ($e2 \% 2$ różne od 0). W takiej sytuacji należy zwrócić wartość mniejszą od 0, jeśli pierwszy argument jest mniejszy, wartość większą od 0, jeśli drugi argument jest mniejszy, oraz wartość 0, jeśli argumenty są równe. Zapewnia to instrukcja `return e1 - e2;`.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 10.1.

Utwórz tablicę przechowującą 50 kolejnych wartości całkowitych i wyświetl jej zawartość na ekranie. W obu operacjach użyj pętli typu `while`.

Ćwiczenie 10.2.

Utwórz tablicę przechowującą w komórkach o indeksach od 0 do 99 kolejne wartości całkowite od 100 do 1 (w porządku malejącym) i wyświetl jej zawartość na ekranie. W obu operacjach użyj pętli typu `for`, a do obliczeń zastosuj tylko zmienną iteracyjną.

Ćwiczenie 10.3.

Utwórz tablicę, która w komórkach o indeksach od 0 do 49 przechowuje wartości od 1 do 50, a w komórkach indeksach od 50 do 99 wartości od 100 do 51 oraz wyświetl jej zawartość na ekranie. Do tworzenia tablicy użyj tylko jednej pętli typu `for`, a do obliczeń wykorzystaj tylko zmienną iteracyjną.

Ćwiczenie 10.4.

Popraw kod z listingu 3.26, tak aby na ekranie pojawiły się jedynie wartości zdefiniowanych komórek tablicy (by nie pojawiały się słowa `undefined`). Zmodyfikuj wyłącznie wnętrze pętli odczytującej dane z tablicy (nie zmieniaj wyrażeń pętli: `i = 0;`, `i < tab.length;`, `i++`), ale też nie używaj instrukcji warunkowej `if`.

Ćwiczenie 10.5.

Wykonaj sortowanie tablicy zawierającej wartości całkowite, tak aby najpierw znalazły się wartości niepodzielne przez 3 w porządku malejącym, a następnie wartości podzielne przez 3 w porządku rosnącym.

Lekcja 11. Obsługa błędów i wyjątki

Wyjątki (z ang. *exceptions*) to konstrukcje programistyczne służące do obsługi sytuacji — jak sama nazwa wskazuje — wyjątkowych. Najczęściej wykorzystywane są do obsługi błędów, pozwalając na zmniejszenie liczby instrukcji warunkowych oraz pomijanie całych bloków kodu, dzięki czemu skrypty są bardziej przejrzyste i łatwiej nimi zarządzać. W JavaScriptie, począwszy specyfikacji ECMAScript v3, również można korzystać z mechanizmu wyjątków. Obsługują je więc wszystkie współczesne przeglądarki. Osoby znające klasyczne języki programowania, takie jak C++ czy Java, nie będą miały żadnych problemów z posługiwaniem się tymi konstrukcjami, gdyż ich składnia jest zapożyczona wprost z tego typu języków.

Zgłaszanie wyjątków

Wyjątek można zgłosić za pomocą instrukcji `throw`. Ma ona postać:

`throw wyrażenie;`

Czynność tę określa się również jako „wyrzucanie” wyjątku (od ang. *throw* — rzucać⁸). Ciąg *wyrażenie* to wyrażenie określające wyjątek; może ono być dowolnego typu, aczkolwiek obecnie zaleca się korzystanie z typu `Error` (zajmiemy się nim już za chwilę).

Zgłoszenie wyjątku to informacja dla interpretera JavaScriptu (mówiąc potocznie: dla przeglądarki), że wystąpiła sytuacja nadzwyczajna, która wymaga specjalnej obsługi. Z reguły jest to informacja o błędzie. Zobaczmy zatem, jak zareaguje przeglądarka na wykonanie instrukcji `throw`. Jako *wyrażenie* użyjemy po prostu ciągu znaków. W pliku `skrypt.js` umieszczaśmy zatem jedną tylko instrukcję:

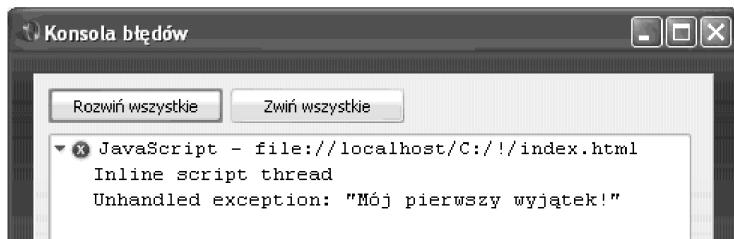
`throw "Mój pierwszy wyjątek!";`

⁸ W języku angielskim stosowany jest również termin *raise* (np.: *raise an exception*).

i wczytujemy kod do przeglądarki. W efekcie zobaczymy pustą stronę. Czy zatem instrukcja nie zadziałała? Zadziałała, jednak informacje o wyjątkach czy błędach nie pojawiają się bezpośrednio na stronie WWW, ale na konsoli błędów, którą udostępnia każda przyzwoita przeglądarka. W Firefoksie dostęp do konsoli otrzymamy, wybierając z menu *Narzędzia* opcję *Konsola błędów* (lub wciskając kombinację klawiszy CTRL+SHIFT+J), a w Operze — wybierając z menu *Narzędzia* pozycje *Zaawansowane* i *Konsola błędów*. W oknie konsoli zobaczymy widok, podobny do przedstawionego na rysunku 3.24.

Rysunek 3.24.

Informacja o wyjątku pojawiła się na konsoli błędów



Komunikat może być różny, w zależności od zastosowanej przeglądarki, zawsze będzie jednak informował o nieobsłużonym (ang. *unhandled*) czy też nieprzechwyconym (ang. *uncaught*) wyjątku. Będzie też wyświetlana wartość wyrażenia użytego w instrukcji `throw` skonwertowana do typu `string`. W tym przypadku jest to ciąg znaków `Mój pierwszy wyjątek!`.

Zamiast jednak stosować w parametrze instrukcji `throw` typy proste, lepiej — i jest to zalecane postępowanie — użyć obiektu opisującego błąd. W przypadku ogólnym jest to obiekt typu `Error` (inne typy omówimy w dalszej części lekcji). Wtedy instrukcja `throw` będzie miała ogólną postać:

`throw new Error();`

lub:

`throw new Error(komunikat)`

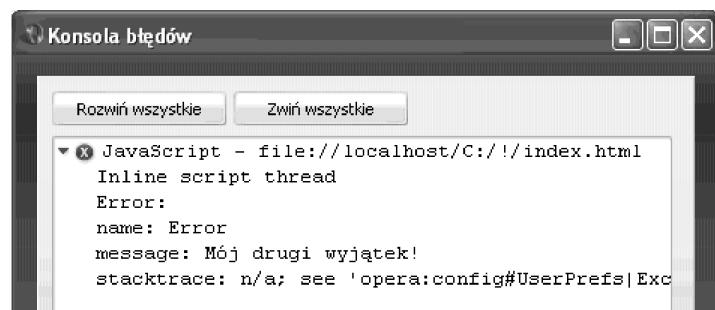
W pierwszym przypadku powstanie pusty obiekt typu `Error`, w drugim — otrzyma on zdefiniowany przez nas komunikat. Obiekt `Error` posiada dwie właściwości, `name` i `message`⁹. Pierwsza z nich określa typ obiektu błędu i faktycznie jest nazwą konstruktora wywołanego do utworzenia tego obiektu. Druga zawiera komunikat przekazany jako argument konstruktora i powinien to być opis wyjątku (błędu). Jeżeli komunikat nie został przekazany, przeglądarka może umieścić swój własny (zwyczek *Generic error*, *General exception* lub podobny). Jeżeli zatem w kodzie umieścimy instrukcję:

`throw new Error("Mój drugi wyjątek!");`

i odświeżymy stronę, w konsoli błędów przeglądarki Opera zobaczymy informacje zaprezentowane na rysunku 3.25.

⁹ Część przeglądarek dodaje również swoje własne, niestandardowe właściwości. Ponieważ jednak są one rozpoznawane wyłącznie przez niektóre produkty, nie będziemy ich omawiać.

Rysunek 3.25.
Wyjątek zgłoszony
za pomocą obiektu
Error



Oprócz właściwości name i message, obiekt typu Error zawiera również metodę `toString`, która zwraca jego opis w postaci ciągu znaków. Postać tego ciągu jest zależna od implementacji i w różnych przeglądarkach wygląda odmienne.

Przechwytywanie wyjątków

Zgłoszenie wyjątku to sygnalizacja wystąpienia sytuacji wyjątkowej, najczęściej jakiegoś błędu. Gdyby jednak możliwości JavaScriptu kończyły się tylko na użyciu instrukcji `throw`, technika ta nie byłaby tak bardzo użyteczna. Z pewnością jednak domyślamy się, że istnieją konstrukcje umożliwiające reakcję na wyjątki. Mówimy wtedy o przechwytywaniu wyjątków, a służy do tego instrukcja `try...catch` o następującej postaci:

```
try{
    //instrukcje mogące spowodować wyjątek
}
catch (identyfikatorWyjątku){
    //kod obsługujący wyjątek
}
```

W nawiasie klamrowym występującym po słowie `try` umieszczać instrukcje, które mogą spowodować zaistnienie wyjątku. W bloku występującym po `catch` umieszczać kod, który ma zostać wykonany, kiedy wyjątek wystąpi. Wyrażenie `identyfikatorWyjątku` możemy traktować jak zmienną wskazującą obiekt wyjątku. Sposób użycia tej konstrukcji zobrazowano w skrypcie z listingu 3.31.

Listing 3.31. Przechwycenie wyjątku

```
var str = "";

str = "Ta strona działa wspaniale!";
try{
    throw new Error("Wyjątkowo poważny błąd!");
}
catch(e){
    str = "Bardzo mi przykro, ale wystąpił błąd.";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Na początku kodu mamy zmienną str zawierającą treść, która ma pojawić się na witrynie. Dalej znajduje się konstrukcja try...catch. W bloku try została umieszczona instrukcja throw, która na pewno spowoduje wystąpienie wyjątku. Po tym, gdy wyjątek powstanie i będzie zgłoszony przez throw, zostanie przechwycony w bloku catch. Tam też zostanie zmieniona treść, która ma się pojawić na witrynie jako komunikat o błędzie. Kiedy uruchomimy taki kod, zobaczymy, że na stronie pojawiła się informacja o nieprawidłowości, natomiast na konsoli błędów nie będzie żadnych komunikatów. To oznacza, że przechwyciliśmy wyjątek i obsłuzyliśmy go samodzielnie, nie było więc konieczności interwencji przeglądarki. Oczywiście, jeśli wyjątek nie powstanie, nie zostanie też wykonany blok catch. Ujmijmy więc w komentarzu instrukcję throw:

```
// throw new Error("Wyjątkowo poważny błąd!");
```

Po odświeżeniu strony okaże się, że pojawił się na niej pierwotny komunikat przypisany zmiennej str.

Nie skorzystaliśmy jednak z identyfikatora wyjątku występującego w bloku catch, a przecież zawiera on obiekt wyjątku powstałego w instrukcji throw. Spróbujmy więc użyć go do pobrania wartości opisanych wcześniej właściwości obiektu Error. Wtedy wyświetlany przez nas tekst będzie mógł zawierać oryginalny komunikat opisujący błąd, a także jego typ. Takie zadanie realizuje skrypt widoczny na listingu 3.32, a efekt jego działania został zaprezentowany na rysunku 3.26.

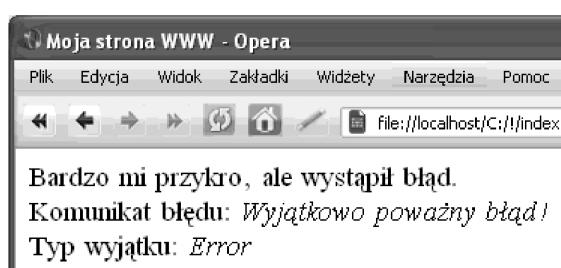
Listing 3.32. Użycie właściwości obiektu Error

```
var str = "";

str = "Ta strona działa wspaniale!";
try{
    throw new Error("Wyjątkowo poważny błąd!");
}
catch(e){
    var komunikat = e.message;
    var typ = e.name;
    str = "Bardzo mi przykro, ale wystąpił błąd.<br />";
    str += "Komunikat błędu: <i>" + komunikat + "</i><br />";
    str += "Typ wyjątku: <i>" + typ + "</i><br />";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Rysunek 3.26.
Wyświetlenie komunikatu i określenie typu błędu



W stosunku do przykładu z listingu 3.31 zmieniła się treść sekcji catch. Tym razem do odczytania typu wyjątku oraz przypisanego mu komunikatu używamy obiektu przekazanego do sekcji catch, a reprezentowanego przez argument (zmienną) e. Typ jest reprezentowany przez właściwość name, a komunikat — przez właściwość message. Odczytujemy zapisane w tych właściwościach dane, przypisujemy je zmiennym pomocniczym typ i komunikat oraz używamy tych zmiennych do skonstruowania treści, która ma się pojawić na stronie.

Obsługa błędów w praktyce

Wyjątków nie zgłaszamy tak sobie, a jedynie w konkretnych sytuacjach. Zastanówmy się więc, czy i kiedy warto z nich skorzystać, i jak inaczej można radzić sobie z obsługą błędów. Rozważmy prostą funkcję przyjmującą dwa argumenty, której zadaniem jest zwrócenie wyniku dzielenia pierwszego przez drugi. Skrypt korzystający z takiej funkcji mógłby wyglądać tak, jak na listingu 3.33.

Listing 3.33. Użycie funkcji dzielącej argumenty

```
var str = "";

function podziel(arg1, arg2)
{
    return arg1 / arg2;
}

var wynik = podziel(8, 0);
str += "Wynikiem jest: " + wynik;

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Funkcja podziel zwraca wynik ilorazu arg1 / arg2. Zwróćmy jednak uwagę, że została wywołana z argumentami 8 i 0. Ósemki przez 0 na pewno nie podzielimy. Jest to operacja niedozwolona. Co będzie wynikiem działania funkcji? Wartość Infinity. Jeżeli jednak użytkownik naszej strony zobaczy na niej tekst: „Wynikiem jest: Infinity”, na pewno nie będzie zadowolony, taki komunikat nie jest bowiem zrozumiały. Nie tylko jednak użytkownikowi może to przysporzyć problemów. Nam, jako autorom skryptu, również. Jeżeli bowiem wynik działania funkcji podziel miałby być używany w dalszych obliczeniach, trudno będzie zorientować się, w którym miejscu powstał błąd. To nie jedyny problem. Zmieńmy wywołanie funkcji np. na:

```
var wynik = podziel(8, "abc");
```

Tym razem na stronie pojawi się komunikat: „Wynikiem jest: NaN”. Nic dziwnego. Wartość abc na pewno nie jest liczbą i działanie nie może zostać wykonane.

Jednym ze sposobów poradzenia sobie z tymi problemami jest badanie wyniku zwróconego przez funkcję. Możemy użyć instrukcji warunkowej if i znanych już funkcji isNaN i isFinite (lekcja 7.). Wtedy fragment kodu:

```
str += "Wynikiem jest: " + wynik;
```

powinniśmy zamienić na:

```
if(isNaN(wynik) || !isFinite(wynik)){
    str += "Obliczenia nie mogły zostać wykonane.";
}
else{
    str += "Wynikiem jest: " + wynik;
}
```

Teraz zachowanie skryptu będzie już dużo lepsze. Dla poprawnych argumentów na ekranie pojawi się wynik, a w przypadku nieprawidłowych — informacja o niemożności wykonania obliczeń. „Duże lepsze” nie znaczy jednak, że satysfakcyjne. Cóż to bowiem znaczy, że obliczenia nie mogły zostać wykonane? Jaka była tego przyczyna? Dzielenie przez 0? Nieprawidłowe argumenty? A jeśli nieprawidłowe argumenty, to który z nich jest błędny? Co więcej, być może wartość Infinity, czyli nieskończoność, jest poprawnym wynikiem, a my z góry taką możliwość odrzuciliśmy. Komunikat powinien być bardziej dokładny. A to oznacza, że trzeba badać stan argumentów w funkcji podziel.

O ile jednak ich zbadanie nie przysporzy na pewno żadnego problemu, o tyle kwestia sygnalizacji błędów nie jest już taka oczywista. Jak bowiem zakomunikować, że argumenty nie są prawidłowe? Możemy skorzystać z właściwości typowej dla wielu języków skryptowych (w tym i JavaScriptu) polegającej na tym, że funkcja może zwrócić wartość dowolnego typu. Postępowanie może być więc następujące: jeżeli argumenty są prawidłowe, funkcja zwróci wartość dzielenia, jeżeli nie są — komunikat określający błąd. Skrypt przyjąłby wtedy postać widoczną na listingu 3.34.

Listing 3.34. Typ wyniku zależny od rezultatu operacji

```
var str = "";

function podziel(arg1, arg2)
{
    var komunikat = "";
    if(isNaN(arg1)){
        komunikat += "Nieprawidłowy pierwszy argument ";
        komunikat += "(arg1 = " + arg1 + ").";
        return komunikat;
    }
    if(isNaN(arg2) || arg2 == 0){
        komunikat += "Nieprawidłowy drugi argument ";
        komunikat += "(arg2 = " + arg2 + ").";
        return komunikat;
    }
    return arg1 / arg2;
}

var wynik = podziel(8, 0);

if(isNaN(wynik)){
    str += "Obliczenia nie mogły zostać wykonane.";
    str += "Przyczyna błędu: <br />";
    str += wynik;
}
```

```
else{
    str += "Wynikiem jest: " + wynik;
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Kod funkcji `podziel` został znacząco rozbudowany, musi ona bowiem badać stan argumentów. Najpierw za pomocą instrukcji warunkowej `if` oraz funkcji `isNaN` bada, czy pierwszy argument stanowi wartość liczbową (a dokładniej, czy może być interpretowany jako wartość liczbowa). Jeśli nie, warunek `isNaN(arg1)` ma wartość `true`, tworzący jest zatem komunikat mówiący o tym, że pierwszy argument jest niewłaściwy. Treść tego komunikatu zawarta w zmiennej pomocniczej `komunikat` jest zwracana jako rezultat działania funkcji, przy użyciu instrukcji `return`.

Jeżeli jednak argument `arg1` jest wartością liczbową, badany jest stan argumentu `arg2`. W tym przypadku badane są dwie rzeczy. Po pierwsze, czy `arg2` nie jest wartością liczbową (`isNaN(arg2)`), po drugie, czy jest równy 0 (`arg2 == 0`). Gdy którykolwiek z tych warunków jest prawdziwy, czyli gdy argument jest nieprawidłowy, podobnie jak w przypadku `arg1` formułowany jest komunikat, który jest zwracany za pomocą instrukcji `return`.

Kiedy oba argumenty są prawidłowe, funkcja zwraca po prostu wynik ich dzielenia:

```
return arg1 / arg2;
```

W dalszej części kodu funkcja `podziel` została wywołana (z parametrami 8 i 0), a rezultat został przypisany zmiennej `wynik`. Możliwe są dwie sytuacje: albo wynik stanowi wartość liczbową, co oznacza, że argumenty były poprawne, albo nie jest wartością liczbową, co oznacza, że wystąpił błąd. W drugim przypadku zmienna `wynik` zawiera komunikat informujący o jego przyczynie. Używając więc instrukcji warunkowej `if` i funkcji `isNaN`, badamy rezultat działania funkcji `podziel` i albo prezentujemy wynik na witrynie, albo wyświetlamy szczegółowy komunikat o błędzie (rysunek 3.27).

Rysunek 3.27.

Wyświetlenie szczegółowego komunikatu o błędzie



Taki sposób sygnalizacji błędów, choć formalnie poprawny, nie jest jednak zalecaną praktyką programistyczną. Będzie budził opór szczególnie wśród osób znających klasyczne języki programowania, w których występuje ściślejsza kontrola typów. Można powiedzieć, że taka funkcja zachowuje się niespójnie, raz zwracając właściwy wynik, a innym razem — komunikat informacyjny. Można by to zmienić, np. w taki sposób, aby zwracała obiekt z rezultatem działania i badać stan tego obiektu (wtedy zawsze

będzie zwracała ten sam typ wyniku), ale przecież możemy skorzystać z techniki wyjątków. Po wykryciu nieprawidłowych argumentów można wygenerować wyjątek, a przy wywoływaniu funkcji tenże wyjątek przechwycić. Jak to zrobić, obrazuje przykład widoczny na listingu 3.35.

Listing 3.35. Wyjątek jako sposób sygnalizacji błędu

```
var str = "";

function podziel(arg1, arg2)
{
    var komunikat = "";
    if(isNaN(arg1) || isNaN(arg2)){
        komunikat += "Nieprawidłowe argumenty ";
        komunikat += "(arg1 = " + arg1 + ". ";
        komunikat += "arg2 = " + arg2 + ").";
        throw new Error(komunikat);
    }
    if(arg2 == 0){
        komunikat += "Niedozwolone dzielenie przez zero ";
        komunikat += arg1 + " / " + arg2 + ").";
        throw new Error(komunikat);
    }
    return arg1 / arg2;
}

try{
    var wynik = podziel(8, 0);
    str += "Wynikiem jest: " + wynik;
}
catch(e){
    str += "Obliczenia nie mogły zostać wykonane.";
    str += "<br />Przyczyna błędu: <br />";
    str += e.message;
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Zasada działania funkcji podziel jest podobna do tej z listingu 3.35, choć dla urozmaicenia został zmieniony sposób badania warunków. Najpierw bowiem sprawdzamy, czy oba argumenty — arg1 i arg2 — stanowią wartości liczbowe. Nie sprawdzamy ich osobno, ale za to w komunikacie uwzględniamy wartości obu. Druga instrukcja warunkowa if bada, czy drugi z argumentów jest równy 0. Jeśli tak, tworzy komunikat o niedozwolonym dzieleniu przez 0. Jednak najważniejsza zmiana w stosunku do kodu z listingu 3.35 to sposób sygnalizacji błędu. Tym razem, jeśli wystąpi jakaś nieprawidłowość, tworzony jest obiekt błędu, któremu w konstruktorze jest przekazywany przygotowany przez nas komunikat, i za pomocą instrukcji throw zostaje zgłoszony wyjątek:

```
throw new Error(komunikat);
```

W związku z takim zachowaniem funkcji podziel, zmianie musiał ulec również sposób jej wywołania. Niezbędne było użycie bloku try...catch. W sekcji try znalazły się dwie instrukcje:

```
var wynik = podziel(8, 0);
str += "Wynikiem jest: " + wynik;
```

Pierwsza to wywołanie funkcji podziel i pobranie jej wyniku, a druga jest utworzeniem komunikatu zawierającego ten wynik. Jeżeli zatem funkcja zadziała prawidłowo, wynik pojawi się na ekranie. Jeśli co najmniej jeden z argumentów będzie błędny (w tym przypadku jest to argument drugi, równy 0), funkcja wygeneruje wyjątek. Skoro powstanie wyjątku, wykonywanie kodu zostanie przerwane, a sterowanie przekazane do sekcji catch. Tym samym nie zostanie wykonana instrukcja:

```
str += "Wynikiem jest: " + wynik;
```

Zamiast niej przetworzone zostaną instrukcje z sekcji catch, a więc na ekranie pojawi się komunikat, że obliczenia nie mogły zostać wykonane, wraz z dokładnym określeniem przyczyny. Powód błędu będzie bowiem opisany we właściwości message obiektu wyjątku e.

Blok finally

Do bloku try możemy dołączyć sekcję finally, która będzie wykonana zawsze, niezależnie od tego, co będzie działało się w bloku try. Schematycznie taka konstrukcja wygląda następująco:

```
try{
    //instrukcje mogące spowodować wyjątek
}
catch(identyfikator){
    //instrukcje sekcji catch
}
finally{
    //instrukcje sekcji finally
}
```

W takim przypadku cała sekcja catch jest opcjonalna, czyli instrukcja może też przyjąć postać:

```
try{
    //instrukcje mogące spowodować wyjątek
}
finally{
    //instrukcje sekcji finally
}
```

W obu przypadkach, zgodnie z tym, co zostało napisane wcześniej, instrukcje sekcji finally są wykonywane zawsze, niezależnie od tego, jakie instrukcje znajdują się w bloku try oraz czy wystąpi w nim wyjątek, czy nie. Obrazujemy to na przykładzie z listingu 3.36.

Listing 3.36. Ilustracja działania sekcji finally

```
var str = "";
function podziel(arg1, arg2)
{
    //treść funkcji podziel
}

try{
    var wynik = podziel(16, 2);
}
catch(e){
    str += "Przechwycenie wyjątku 1 <br />";
}
finally{
    str += "Sekcja finally 1 <br />";
}
try{
    wynik = obj.podziel(10, 0);
}
catch(e){
    str += "Przechwycenie wyjątku 2 <br />";
}
finally{
    str += "Sekcja finally 2 <br />";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

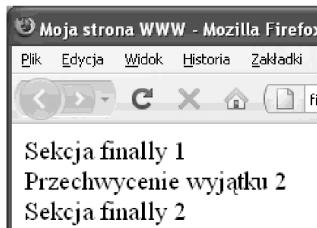
Kod zawiera funkcję podziel, wykonującą dzielenie przekazanych jej argumentów, w postaci znanej z listingu 3.35, dlatego też na listingu 3.36 jej treść została pominięta. Funkcja ta za pomocą instrukcji throw zgłasza wyjątek, gdy rozpozna nieprawidłowe wartości argumentów. Dalsza część kodu ma już inną postać. Funkcja podziel została wywołana dwukrotnie, a oba wywołania ujęto w bloki try...catch...finally. Pierwsze wywołanie ma poprawne argumenty (16 i 2), nie powoduje więc powstania wyjątku, i nie jest też wykonywany pierwszy blok catch. Jest za to wykonywany pierwszy blok finally, gdyż jest niezależny od tego, co znajduje się w sekcji try. Tym samym w oknie przeglądarki najpierw pojawi się napis Sekcja finally 1.

Drugie wywołanie metody podziel powoduje już wygenerowanie wyjątku, bowiem wartość drugiego argumentu jest nieprawidłowa (równa 0). Zostaną zatem wykonane zarówno instrukcje bloku catch, jak i finally. W oknie pojawią się więc dwa kolejne napisy: Przechwycenie wyjątku 2 oraz Sekcja finally 2. Ostatecznie wynik działania całego skryptu będzie taki jak ten, który zaprezentowano na rysunku 3.28.

Jak wspomiano wyżej, sekcję finally można zastosować również w instrukcjach, które nie powodują wygenerowania wyjątku. Działanie jest takie samo jak w przypadku bloku try...catch...finally, tzn. kod z bloku finally zostanie wykonany zawsze, niezależnie od tego, jakie instrukcje znajdują się w bloku try. Jeśli nawet w bloku try znajdzie się instrukcja return lub zostanie wygenerowany wyjątek, blok finally i tak zostanie wykonany. Widać to na przykładzie zaprezentowanym na listingu 3.37.

Rysunek 3.28.

Blok finally jest wykonywany niezależnie od tego, czy pojawi się wyjątek, czy nie

**Listing 3.37. Sekcja finally i instrukcja return**

```
function f()
{
    try{
        return;
    }
    finally{
        alert("Kod sekcji finally został wykonany!");
    }
}
f();
```

Mamy tu funkcję f o maksymalnie uproszczonej konstrukcji. Wywołuje ona instrukcję return. To, jak wiemy, powinno spowodować natychmiastowe zakończenie działania funkcji. A więc żadne instrukcje znajdujące się za return nie powinny zostać wykonane. Jeśli jednak uruchomimy kod, na ekranie zobaczymy okno dialogowe z informacją, że został wykonany kod sekcji finally. Jest to zachowanie prawidłowe, bowiem instrukcja return została ujęta w blok try...finally. Skoro tak, instrukcje znajdujące się w bloku finally musiały zostać wykonane, zanim zadziałała instrukcja return opuszczająca blok funkcji. To najlepszy dowód, że sekcja finally jest wykonywana zawsze, niezależnie od tego, co znajduje się w bloku try.

Użyjmy jednak takiej konstrukcji w bardziej praktycznym przykładzie, zaprezentowanym na listingu 3.38.

Listing 3.38. Użycie sekcji finally

```
var tab = [1, 2, 'a', 'b', 5, 6];
var suma = 0;
var i = 0;

while(i < tab.length){
    try{
        if(isNaN(tab[i])){
            continue;
        }
        suma += tab[i];
    }
    finally{
        i++;
    }
}

alert("Suma komórek tablicy to " + suma);
```

Powyższy skrypt zajmuje się sumowaniem wartości zapisanych w tablicy tab. Czynność ta jest wykonywana w pętli while, której konstrukcja odbiega jednak od dotychczas przez nas stosowanych. Zmienną kontrolującą przebiegi pętli jest i, której wartością początkową jest 0. Warunkiem zakończenia pętli jest $i < \text{tab.length}$, a więc stan zmiennej i zmienia się od 0 do $\text{tab.length} - 1$, odczytywane są więc wszystkie komórki. Sumowanie ich wartości odbywa się za pomocą instrukcji:

```
suma += tab[i];
```

Oznacza to, że w każdym przebiegu do zmiennej suma jest dopisywana wartość komórki wskazywanej przez aktualną wartość i. Trzeba jednak wziąć pod uwagę, że niektóre komórki mogą nie zawierać wartości liczbowych. Wtedy sumowanie nie ma sensu. Przy użyciu instrukcji:

```
if(isNaN(tab[i])){
```

badamy więc, czy `tab[i]` jest wartością numeryczną. Jeśli nie, wykonujemy instrukcję continue, co powoduje rozpoczęcie kolejnego przebiegu pętli¹⁰. Takie zachowanie mogłoby jednak spowodować spory problem. Co by się stało, gdyby pominąć blok try...finally, a instrukcje wewnętrzna pętli wyglądały tak:

```
if(isNaN(tab[i])){
    continue;
}
suma += tab[i];
i++;
```

Odpowiedź jest jedna: gdyby którakolwiek komórka tablicy tab nie zawierała wartości liczbowej, pętla byłaby nieskończona (wykonawałaby się w nieskończoność). W tym przypadku pierwszą komórką niezawierającą liczby jest ta o indeksie 2. Jej odczyt nastąpi zatem, gdy i równe jest 2. Wtedy warunek `isNaN(tab[i])` będzie prawdziwy i zostanie wykonana instrukcja continue. Nastąpi więc kolejny przebieg pętli, jednak i NIE ZMIENI swojej wartości — wciąż będzie równe 2. To spowoduje, że warunek `isNaN(tab[i])` będzie prawdziwy, zostanie więc wykonana instrukcja continue, która rozpocznie kolejny przebieg pętli itd., itd. Taka pętla nie zakończy się nigdy.

Aby nie dopuścić do opisanej sytuacji, zastosowany został blok try...finally. Skoro bowiem instrukcje umieszczone w sekcji finally są wykonywane zawsze, niezależnie od tego, co zdarzy się w sekcji try, to zawsze też zostanie wykonana instrukcja i++ zwiększająca i o 1. Nie ma przy tym znaczenia, czy wcześniej zostanie wykonana instrukcja continue, czy też nie (podobna sytuacja miała miejsce z instrukcją return w kodzie z listingu 3.37). Pętla zakończy się więc dopiero wtedy, gdy fałszywy będzie warunek $i < \text{tab.length}$.

¹⁰ Oczywiście, ten problem można rozwiązać na inne, nawet prostsze, sposoby. Nie mielibyśmy jednak wtedy możliwości przećwiczenia użycia bloku try...finally.

Zagnieżdżanie bloków try...catch

Bloki try...catch można zagnieżdzać. Znaczy to, że w jednym bloku przechwytyującym wyjątek X może istnieć drugi blok, który będzie przechwytywał wyjątek Y. Schematycznie taka konstrukcja wygląda następująco:

```
try{  
    //instrukcje mogące spowodować wyjątek 1  
    try{  
        //instrukcje mogące spowodować wyjątek 2  
    }  
    catch (identyfikatorWyjątku2){  
        //obsługa wyjątku 2  
    }  
}  
catch (identyfikatorWyjątku1){  
    //obsługa wyjątku 1  
}
```

Zagnieżdżenie takie może być wielopoziomowe, czyli w już zagnieżdżonym bloku try można umieścić kolejny taki blok. W praktyce takich piętrowych konstrukcji zazwyczaj się nie stosuje, zwykle nie ma bowiem takiej potrzeby. Maksymalny poziom bezpośredniego zagnieżdżenia z reguły nie przekracza dwóch poziomów.

Propagacja wyjątków

W chwili powstania wyjątku wstrzymywane jest wykonywanie kodu skryptu, a sterowanie zostaje przekazane do najbliższego bloku obsługi wyjątku (z ang. *exception handler*). Jeżeli na danym poziomie hierarchii kodu taki blok nie występuje, sterowanie przekazywane jest do poziomu wyższego. Przypomnijmy sobie kod z listingu 3.35. Występowała tam funkcja podziel, która w pewnych sytuacjach generowała wyjątek. Jednak nigdzie w tej funkcji nie było bloku try...catch, a więc bloku obsługi wyjątku. Skoro tak, wyjątek był przekazywany wyżej, czyli do miejsca wywołania tej funkcji. Tam już był blok try...catch i w nim wyjątek był obsługiwany. Po obsłudzeniu wyjątku wykonywanie kodu skryptu było kontynuowane. Może jednak zdarzyć się tak, że wyjątek będzie wędrował przez wszystkie możliwe szczeble hierarchii kodu (np. przez kolejne zagnieżdżone wywołania funkcji), ale nigdzie nie trafi na blok obsługi. Wtedy skrypt definitelynie zakończy działanie, a informacja pojawi się na konsoli błędów. Dlatego, aby uniknąć awarii skryptu, zawsze należy pamiętać o odpowiedniej obsłudze sytuacji wyjątkowych. Gdy korzystamy z różnych funkcji czy metod, warto sprawdzić, czy i jakie wyjątki generują i, jeśli będzie trzeba, użyć odpowiednich procedur obsługi.

Predefiniowane obiekty wyjątków

Oprócz typu Error, którego używaliśmy w przykładach z tej lekcji, ECMAScript v3 definiuje jeszcze 6 innych: EvalError, RangeError, ReferenceError, SyntaxError, TypeError i URIError. Są dostępne, począwszy od JavaScriptu 1.5. Sposób tworzenia obiektów tych typów jest taki sam jak obiektu Error.

- ◆ EvalError może być zgłaszany w przypadku nieprawidłowego użycia funkcji eval.
- ◆ RangeError może być zgłaszany, gdy wartość numeryczna przekracza dopuszczalny zakres.
- ◆ ReferenceError może być zgłaszany przy próbie odczytu nieistniejących zmiennych.
- ◆ SyntaxError może być zgłaszany po wykryciu błędu składniowego (syntaktycznego), np. przez metodę eval oraz konstruktory Function i RegExp.
- ◆ TypeError może być zgłaszany, gdy wartość jest typu innego niż oczekiwany. Może tak być przy próbie dostępu do właściwości o wartości null bądź undefined, użycia operatora new i argumentu niebędącego konstruktorem, próbie wywołania nieistniejącej metody obiektu itp.
- ◆ URIError może być zgłaszany przez metody kodujące bądź dekodujące adresy URI, gdy wykryte zostaną nieprawidłowo sformowane dane.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 11.1.

Zmodyfikuj kod z listingu 3.35 tak, aby w instrukcji throw zamiast obiektu typu Error był zwracany Twój własny obiekt wyjątku zawierający kod błędu (dowolnie wymyszony) oraz komunikat. Nie zapomnij o utworzeniu prawidłowego konstruktora dla takiego obiektu.

Ćwiczenie 11.2.

Utwórz funkcję przyjmującą dwa argumenty liczbowe. Funkcja powinna zwracać wartość będącą wynikiem odejmowania argumentu pierwszego od drugiego. Jednak w przypadku, gdyby wynik był ujemny, powinien zostać zgłoszony wyjątek. Napisz kod testujący działanie funkcji.

Ćwiczenie 11.3.

Zmodyfikuj kod ćwiczenia 9.5 z lekcji 9. tak, by przy próbie utworzenia obiektu prostokąta o ujemnej szerokości bądź wysokości generowany był odpowiedni wyjątek. Jeżeli w samodzielnym rozwiązaniu ćwiczenia zastosowałeś(aś) inną reprezentację obiektu (bez szerokości i wysokości), użyj rozwiązania z ćwiczenia 9.5. dołączonego do książki.

Rozdział 4.

Współpraca z przeglądarką

Lekcja 12.

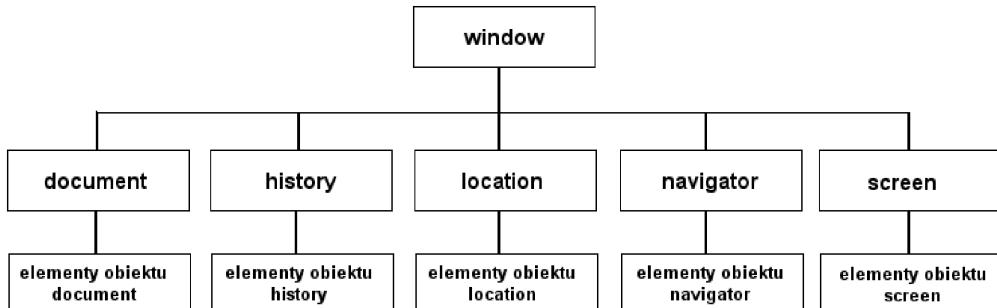
DOM — współpraca z przeglądarką

Każda witryna składa się z szeregu elementów. To przyciski, pola wyboru, akapity tekstowe, warstwy, obrazy itp. Każdy z tych elementów to umieszczony w witrynie osobny obiekt, na którym można wykonywać różne operacje. Oprócz tego, mamy do dyspozycji obiekty główne związane z całym dokumentem i przeglądarką. Aby można było wszystkim w sensowny sposób zarządzać, niezbędne było stworzenie hierarchii obiektów i standaryzacji ich zachowań. Powstał model dokumentu zwany *Document Object Model* — w skrócie DOM. Określa on obiekty, ich właściwości i metody, sposób obsługi zdarzeń itp. Każdą witrynę można więc przedstawić jako hierarchiczną strukturę zgodną z modelem dokumentu stosowanym w danej przeglądarce. Standaryzacją modelu DOM zajęła się organizacja W3C (World Wide Web Consortium — <http://www.w3.org>), ale poszczególne przeglądarki w różnym stopniu respektują jej zalecenia.

W lekcji 12. zajmiemy się standardowymi obiektami najwyższego poziomu (obiektami głównymi, z ang. *top-level objects*), służącymi do kontroli środowiska, w którym uruchamiana jest witryna, czyli do kontroli zachowań przeglądarek, natomiast właściwy model dokumentu dotyczący struktury witryny omówimy w lekcji 13.

Obiekty główne przeglądarki

Obiekty związane z przeglądarką tworzą strukturę hierarchiczną, na szczycie której znajduje się obiekt `window`. Reprezentuje on okno przeglądarki, a jego elementami są inne obiekty, np. `document`, który reprezentuje dokument (X)HTML. Uproszczony schemat tej hierarchii przedstawiono na rysunku 4.1. Przyjrzyjmy się nieco bliżej widniejącym na nim obiektom.



Rysunek 4.1. Uproszczony schemat hierarchii obiektów głównych

Obiekt `window`

Obiekt `window` znajduje się na szczycie hierarchii obiektów i reprezentuje okno przeglądarki. Jest też obiektem domyślnym, tzn. do jego metod i właściwości można się odwoływać bezpośrednio, pomijając jego nazwę. Dokładniej rzecz ujmując, gdy JavaScript operuje w przeglądarce WWW, `window` reprezentuje opisany wcześniej obiekt globalny.

I tak zastosowana na początku kursu instrukcja `alert("tekst")`, wyświetlająca okno dialogowe ze zdefiniowanym tekstem, to nic innego jak metoda obiektu `window`; zatem wywołanie mogłoby również wyglądać następująco:

```
window.alert("tekst");
```

Dokładnie tak samo jest z pozostałymi składowymi tego obiektu. Właściwości obiektu `window` zebrane w tabeli 4.1. W kolumnie *Dostępność* zostały użyte następujące oznaczenia:

FF — właściwość występuje w przeglądarce Firefox,

IE — właściwość występuje w przeglądarce Internet Explorer,

OP — właściwość występuje w przeglądarce Opera,

W3C — właściwość zgodna ze standardami W3C.

Tabela 4.1. Właściwości obiektu window

Nazwa	Znaczenie	Dostępność
closed	Ustawiona na true oznacza, że okno zostało zamknięte.	FF, IE, OP, W3C
defaultStatus	Domyślny tekst wyświetlany na pasku stanu.	FF, IE, OP, W3C
document	Obiekt document zawierający elementy wyświetlanej witryny.	FF, IE, OP, W3C
event	Zawiera odniesienie do obiektu typu Event opisującego ostatnie zdarzenie. Używany wyłącznie w modelu zdarzeń przeglądarki Internet Explorer.	IE
frames	Tablica zawierająca odniesienia do ramek danego okna.	FF, IE, OP
history	Obiekt typu History zawierający historię odwiedzin.	FF, IE, OP, W3C
innerWidth	Wewnętrzna wysokość obszaru okna.	OP, FF
innerHeight	Wewnętrzna szerokość obszaru okna.	OP, FF
length	Liczba ramek potomnych zawartych w oknie.	FF, IE, OP, W3C
location	Obiekt typu Location zawierający URL aktualnego dokumentu.	FF, IE, OP, W3C
locationbar	Obiekt pozwalający na określanie, czy ma być widoczny pasek adresu. Posiada właściwość visible, której można przypisywać wartości true i false.	FF
menubar	Obiekt pozwalający na określanie, czy ma być widoczny pasek menu. Posiada właściwość visible, której można przypisywać wartości true i false.	FF
name	Nazwa bieżącego okna.	FF, IE, OP, W3C
opener	Jeżeli bieżące okno zostało otwarte za pomocą metody open, właściwość opener zawiera odniesienie do okna źródłowego.	NN, OP, FF, W3C
outerHeight	Zewnętrzna wysokość obszaru okna.	OP, FF, W3C
outerWidth	Zewnętrzna szerokość obszaru okna.	OP, FF, W3C
pageXOffset	Przesunięcie bieżącej strony w poziomie względem lewego górnego rogu okna (szerokość zaslonionego obszaru).	OP, FF, W3C
pageYOffset	Przesunięcie bieżącej strony w pionie względem lewego górnego rogu okna (wysokość zaslonionego obszaru).	OP, FF, W3C
parent	Odniesienie do okna źródłowego (nadrzędnego).	FF, IE, OP, W3C
personalbar	Obiekt pozwalający na określanie, czy ma być widoczny pasek ustawień osobistych. Posiada właściwość visible, której można przypisywać wartości true i false.	FF
scrollbars	Obiekt pozwalający na określanie, czy mają być widoczne paski przewijania. Posiada właściwość visible, której można przypisywać wartości true i false.	FF
self	Odniesienie do bieżącego aktywnego okna lub ramki.	FF, IE, OP, W3C
status	Tekst wyświetlany na pasku stanu.	FF, IE, OP, W3C
statusbar	Obiekt pozwalający na określanie, czy ma być widoczny pasek stanu. Posiada właściwość visible, której można przypisywać wartości true i false.	FF

Tabela 4.1. Właściwości obiektu window (ciąg dalszy)

Nazwa	Znaczenie	Dostępność
toolbar	Obiekt pozwalający na określanie, czy ma być widoczny pasek narzędziowy. Posiada właściwość visible, której można przypisywać wartości true i false.	FF
top	Odniesienie do okna znajdującego się najwyżej w hierarchii.	FF, IE, OP, W3C
window	Odniesienie do bieżącego okna lub ramki.	FF, IE, OP

Jeśli chcemy odczytać wszystkie właściwości obiektu window, możemy z powodzeniem użyć pętli for...in poznanej w lekcji 8., w rozdziale 3. Sposób wykonania zawarto w skrypcie z listingu 4.1.

Listing 4.1. Odczyt właściwości obiektu window

```
var str = "";

for(nazwa in window){
    str += "window." + nazwa + " = ";
    try{
        str += window[nazwa] + "<br />"
    }
    catch(e){
        str += "brak odczytu<br />"
    }
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Pętla została użyta w sposób klasyczny, pod zmienną nazwa w każdym przebiegu jest podstawiana wartość kolejnej właściwości. Na tej podstawie konstruowane są kolejne ciągi znaków, zgodne ze schematem:

window.nazwa_właściwości = wartość_właściwości

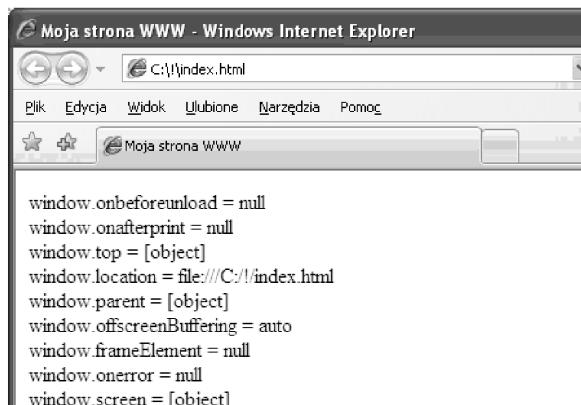
Ciągi te są dodawane do zmiennej str. Dzięki temu pojawią się w oknie przeglądarki. Dodatkowo została tu użyta instrukcja try...catch przechwytyująca ewentualne wyjątki, które mogłyby powstać, gdyby odczyt danej właściwości nie był możliwy (z taką sytuacją spotkamy się np. w przeglądarce Opera). Wtedy tworzony ciąg znaków przyjmie postać:

window.nazwa_właściwości = brak odczytu

Przykładowy efekt działania skryptu w przeglądarce Internet Explorer 7 jest widoczny na rysunku 4.2. Warto jednak uruchomić skrypt w innych przeglądarkach i porównać wyniki. Okaże się, że znajdziemy sporo różnic i niestandardowych właściwości. Niestety, każdy producent dodaje do swojego produktu własne rozszerzenia, najczęściej niezgodne z innymi. Warto też zwrócić uwagę, że wyświetcone zostały wszystkie składowe obiektu window, również metody. Przyjrzyjmy się im bliżej.

Rysunek 4.2.

Odczytanie właściwości obiektu window w przeglądarce Internet Explorer 7

**Metody obiektu window**

Obiekt window został wyposażony w spory zestaw metod pozwalających na wykonywanie rozmaitych zadań. Niektóre z nich mogą się bardzo przydać przy pisaniu skryptów. Metody udostępniane przez obiekt window zostały zebrane w tabeli 4.2. Jak widać, część z nich jest dostępna jedynie w wybranych przeglądarkach. Skupimy się jednak na funkcjach standardowych, których działanie nie zależy od danej implementacji.

Tabela 4.2. Wybrane metody obiektu window

Metoda	Znaczenie	Dostępność
alert("str")	Wyświetla okno dialogowe zawierające tekst str.	IE, FF, OP
back()	Odpowiada wciśnięciu przycisku <i>Back (Wstecz)</i> w przeglądarce.	FF, OP
blur()	Usuwa fokus z bieżącego okna.	IE, FF, OP
captureEvents(typ)	Ustala, że bieżące okno ma przechwytywać zdarzenia typu typ.	FF, OP
clearInterval(id)	Zatrzymuje timer id uruchomiony przez metodę setInterval.	IE, FF, OP
clearTimeout(id)	Zatrzymuje timer id uruchomiony przez metodę setTimeout.	IE, FF, OP
close()	Zamyka okno.	IE, FF, OP
confirm("str")	Wyświetla okno dialogowe z tekstem str oraz przyciskami <i>OK</i> i <i>Cancel</i> .	IE, FF, OP
disableExternalCapture()	Wyłącza przechwytywanie zdarzeń, włączone za pomocą metody enableExternalCapture.	FF, OP
enableExternalCapture()	W oknie posiadającym ramki włącza przetwarzanie zdarzeń w dokumentach pochodzących z serwerów zewnętrznych.	FF, OP
find([str[, caseSensitive, ↵backward]])	Umożliwia przeszukanie treści bieżącego okna pod kątem występowania ciągu str. Jeżeli nie zostanie podany żaden argument, przeglądarka wyświetli okno dialogowe umożliwiające wprowadzenie danych. Parametr caseSensitive ustawiony na true oznacza uwzględnianie wielkości liter, natomiast backward ustawiony na true oznacza przeszukiwanie od końca dokumentu.	FF

Tabela 4.2. Wybrane metody obiektu window (ciąg dalszy)

Metoda	Znaczenie	Dostępność
focus()	Ustawia fokus na bieżącym oknie.	IE, FF, OP
forward()	Odpowiada wciśnięciu przycisku <i>Forward (W przód)</i> w przeglądarce.	FF, OP
handleEvent(eventId)	Umożliwia wywołanie procedury obsługi zdarzenia określonego przez parametr eventId.	FF, OP
home()	Odpowiada wciśnięciu przycisku <i>Home (Start, Główna itp.)</i> w przeglądarce.	FF, OP
moveBy(px, py)	Przesuwa okno o px pikseli w poziomie i py pikseli w pionie, względem aktualnego położenia.	IE, FF, OP
moveTo(px, py)	Przesuwa okno tak, aby jego lewy górny róg znalazł się w punkcie px, py.	IE, FF, OP
open(URL, nazwa ↳[, właściwości ↳[, zamiana]])	Otwiera nowe okno o nazwie <i>nazwa</i> , z dokumentem wskazywanym przez URL. Wygląd okna określa parametr <i>właściwości</i> .	IE, FF, OP
print()	Drukuję zawartość okna.	IE, FF, OP
prompt(str[, defStr])	Wyświetla okno dialogowe z komunikatem str, zawierające pole tekstowe umożliwiające wprowadzenie danych. W polu tekstowym wyświetlony zostanie ciąg defStr.	IE, FF, OP
releaseEvents(typ)	Zwalnia przechwycone wcześniej zdarzenia o typie wskazywanym przez typ.	FF, OP
resizeBy(px, py)	Przemieszcza prawy dolny róg okna do punktu o współrzędnych px, py.	IE, FF, OP
resizeTo ↳(szerokość, wysokość)	Zmienia rozmiary okna.	IE, FF, OP
routeEvent(typ)	Przekazuje obsługę zdarzenia do kolejnego obiektu w hierarchii.	FF
scroll(x, y)	Przesuwa zawartość okna do współrzędnych x, y. Metoda przestarzała — o ile to możliwe, należy korzystać z metody scrollTo.	IE, FF, OP
scrollBy(px, py)	Przesuwa zawartość okna o px pikseli w poziomie i py pikseli w pionie. Metoda użyteczna tylko w przypadku, kiedy zawartość dokumentu nie mieści się w oknie.	IE, FF, OP
scrollTo(x, y)	Przesuwa zawartość okna do współrzędnych x, y.	IE, FF, OP
setInterval(exp, mtime)	Przetwarza wyrażenie (lub wywołuje funkcję) exp po czasie mtime milisekund.	IE, FF, OP
setTimeout(exp, mtime)	Przetwarza wyrażenie (lub wywołuje funkcję) exp po czasie mtime milisekund.	IE, FF, OP
stop()	Odpowiada wciśnięciu przycisku <i>Stop</i> w przeglądarce.	FF, OP

Metodę alert znamy już doskonale. Wyświetla okno dialogowe zawierające tekst przekazany jako argument. Zobaczmy zatem, jak wyświetlić okno dialogowe innego typu, np. takie, które pozwoli na zaakceptowanie lub odrzucenie jakiejś operacji. W tym celu wystarczy użyć metody `confirm`. Jej działanie jest bardzo proste. Powoduje wyświetlenie okna dialogowego zawierającego przyciski *OK* i *Anuluj (Cancel)*. Postać tego okna będzie podobna we wszystkich przeglądarkach. W Internet Explorerze będzie wyglądało tak, jak na rysunku 4.3. Oczywiście, samo wyświetlenie okna to nie wszystko. Metoda pozwala też stwierdzić, który z przycisków został kliknięty. Jeśli bowiem zwróconą wartością będzie true, oznacza to, że kliknięto *OK*, jeśli będzie to false — kliknięto *Anuluj*. Użycie tej metody w praktyce zobrazowano w skrypcie z listingu 4.2.

Rysunek 4.3.
Okno dialogowe
pozwalające
na potwierdzenie
operacji



Listing 4.2. Użycie metody `confirm` do potwierdzenia operacji

```
var str = "";

if(confirm("Czy chcesz dokonać zakupu?")){
    str += "Towar został dodany do koszyka.";
}
else{
    str += "Transakcja została anulowana.";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Instrukcja `if` bada tu wartość zwróconą przez metodę `confirm`. W zależności od tego, czy jest to true (kliknięto przycisk *OK*), czy false (kliknięto przycisk *Anuluj*), wyświetla na ekranie odpowiedni tekst. Wywołanie `confirm` zostało umieszczone bezpośrednio w instrukcji warunkowej. Dla zwiększenia czytelności kodu można by też użyć dodatkowej zmiennej pomocniczej i jej wartość badać w `if`:

```
var rezultat = confirm("Czy chcesz dokonać zakupu?");
if(rezultat){
    /* dalsza część kodu */
```

Inną ciekawą metodą jest `prompt`. Wyświetla okno dialogowe, które pozwala użytkownikowi strony na wprowadzenie danych (ciagu znaków). Wprowadzony ciąg znaków jest zwracany jako wynik działania funkcji. Kiedy nie zostaną wprowadzone żadne dane bądź w oknie zostanie kliknięty przycisk *Anuluj (Cancel)*, wynikiem działania będzie wartość `null`. Funkcji można przekazać dwa argumenty: pierwszy określa, jaki tekst zostanie wyświetlony w oknie (np. treść pytania, które chcemy zadać użytkownikowi), natomiast drugi pozwala na zdefiniowanie domyślnego ciągu znaków

umieszczonego w polu tekstowym wyświetlanego okna dialogowego. Jeśli drugi argument nie zostanie podany, domyślnym ciągiem w większości przypadków będzie pusty串 znaków (natomiast w przeglądarce Internet Explorer串 undefined). Jak w praktyce działa funkcja prompt, można zobaczyć, czytając skrypt z listingu 4.3.

Listing 4.3. Wprowadzanie danych przez użytkownika

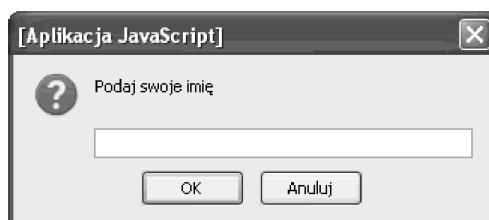
```
var str = "";
var imię = prompt("Podaj swoje imię", "");

if(imię == null || imię == ""){
    str += "Szkoda, że nie chcesz podać swojego imienia.";
}
else{
    str += "Cześć " + imię;
    str += ". Witamy na naszej stronie.";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

W drugiej linii skryptu jest wywoływana metoda prompt wyświetlająca okno dialogowe ze zdefiniowanym tekstem, zawierające również pole tekstowe pozwalające na wprowadzanie danych. Okno to będzie miało nieco inny wygląd, w zależności od zastosowanej przeglądarki (rysunki 4.4 i 4.5). Drugim argumentem jest pusty串 znaków "", dzięki czemu w przeglądarce Internet Explorer unikniemy wyświetlenia w polu tekstowym okna dialogowego ciągu undefined.

Rysunek 4.4.
Okno dialogowe
metody prompt
w przeglądarce
Firefox



Rysunek 4.5.
Okno dialogowe
metody prompt
w przeglądarce
Internet Explorer



Wynik działania metody prompt jest przypisywany zmiennej imię. Będzie to:

- ◆串 znaków wprowadzony przez użytkownika, jeśli wpisał dane do pola tekstowego i kliknął OK;
- ◆ pusty串 znaków, jeśli użytkownik nie wpisał żadnych danych i kliknął OK;
- ◆ wartość null, jeśli użytkownik kliknął Anuluj (Cancel).

Jeśli zatem zmienna `imię` ma wartość `null` lub zawiera pusty ciąg znaków (zapisywany jako `" "`), oznacza to, że żadne dane nie zostały wprowadzone. Wtedy na ekranie pojawi się napis:

Szkoda, że nie chcesz podać swojego imienia.

Jeżeli jednak zmienna `imię` będzie zawierała inne dane, jej wartość zostanie użyta do skonstruowania komunikatu powitalnego, np.:

Cześć Magda. Witamy na naszej stronie.

Oczywiście, skrypt ten stanowi tylko ilustrację możliwości wykorzystania metody `prompt`. Nie należy go raczej umieszczać na realnie działającej witrynie, chyba że chcemy ziryutować naszych użytkowników.

Ostatnią metodą, którą omówimy w tej części lekcji, jest `open`. Pozwala otworzyć nowe okno przeglądarki, o ile taka możliwość nie została zablokowana w opcjach konfiguracyjnych danego produktu. Wywołanie `open` ma postać:

```
open(URL, nazwa [, właściwości[, zamiana]])
```

Nowe okno będzie miało nazwę `nazwa` i zawierało dokument wskazywany przez `URL`. Wygląd okna można określić za pomocą opcjonalnego argumentu `właściwości`, który może przyjmować parametry przedstawione w tabeli 4.3. Poszczególne parametry należy oddzielać od siebie znakami przecinka. Opcjonalny argument `zamiana` ustawiony na `true` określa, że otwierany dokument ma się pojawić w historii otwieranych witryn jako nowy wpis, a ustawiony na `false`, że ma zamienić aktualny wpis. Metoda `open` zwraca obiekt wskazujący nowe okno, pozwalający na wykonywanie na nim innych operacji (wywoływanie metod, zmiana właściwości itp.).

Tabela 4.3. Właściwości określające wygląd okna otwieranego za pomocą metody `open`

Właściwość	Znaczenie	Dopuszczalne wartości	Przykład	Dostępność
<code>left</code>	Współrzędna x lewego górnego rogu okna.	liczby całkowite	<code>left=100</code>	FF, IE, OP
<code>top</code>	Współrzędna y lewego górnego rogu okna.	liczby całkowite	<code>top=10</code>	FF, IE, OP
<code>height</code>	Wysokość obszaru okna zawierającego treść strony, włącznie z wysokością poziomego paska przewijania.	liczby całkowite, minimalna wartość: 100	<code>height=200</code>	FF, IE, OP
<code>width</code>	Szerokość obszaru okna zawierającego treść strony, włącznie z szerokością pionowego paska przewijania.	liczby całkowite, minimalna wartość: 100	<code>width=200</code>	FF, IE, OP
<code>outerHeight</code>	Zewnętrzna wysokość całego obszaru okna, włącznie z wszystkimi paskami narzędziowymi.	liczby całkowite, minimalna wartość: 100	<code>outerHeight ↴=200</code>	FF
<code>outerWidth</code>	Zewnętrzna szerokość całego obszaru okna, włącznie z wszystkimi paskami narzędziowymi.	liczby całkowite, minimalna wartość: 100	<code>outerWidth ↴=200</code>	FF

Tabela 4.3. Właściwości określające wygląd okna otwieranego za pomocą metody open (ciąg dalszy)

Właściwość	Znaczenie	Dopuszczalne wartości	Przykład	Dostępność
innerHeight	Wysokość obszaru okna zawierającego treść strony (odpowiednik height), włącznie z wysokością poziomego paska przewijania.	liczby całkowite, minimalna wartość: 100	innerHeight ↳=200	FF
innerWidth	Szerokość obszaru okna zawierającego treść strony (odpowiednik width), włącznie z szerokością pionowego paska przewijania.	liczby całkowite, minimalna wartość: 100	innerWidth ↳=200	FF
menubar	Określa, czy ma być widoczny pasek menu.	yes, no	menubar=yes	FF, IE
toolbar	Określa, czy ma być widoczny pasek narzędziowy (nawigacyjny).	yes, no	toolbar=yes	FF, IE
location	Określa, czy ma być widoczny pasek adresu.	yes, no	location ↳=yes	FF, IE, OP
directories	Określa, czy ma być widoczny pasek ustawień osobistych (Personal Toolbar). Jego zawartość zależy od konkretnej przeglądarki.	yes, no	directories ↳=yes	FF, IE, OP
personalbar	Określa, czy ma być widoczny pasek ustawień osobistych (Personal Toolbar). Odpowiednik opcji directories. Jego zawartość zależy od konkretnej przeglądarki.	yes, no	personalbar ↳=yes	FF
status	Określa, czy ma być widoczny pasek stanu (statusu).	yes, no	status=yes	FF, IE, OP
resizable	Określa, czy wymiary okna mogą być zmieniane.	yes, no	resizable ↳=yes	FF, IE
scrollbars	Określa, czy w przypadku, kiedy dokument nie mieści się w oknie, mają być wyświetlane paski przewijania.	yes, no	scrollbars ↳=yes	FF, IE
dependent	Określa, czy nowe okno ma być zależne od okna otwierającego.	yes, no	dependent ↳=yes	FF
chrome	Pozwala otworzyć okno nieposiadające żadnego interfejsu.	yes, no	chrome=yes	FF
modal	Określa, czy okno ma być modalne.	yes, no	modal=yes	FF
dialog	Określa, czy okno ma być tworzone jako dialogowe.	yes, no	dialog=yes	FF
minimizable	Pozwala określić, czy okno dialogowe ma posiadać przycisk minimalizacji.	yes, no	minimizable ↳=yes	FF

Jeśli zatem w skrypcie użyjemy instrukcji:

```
window.open("http://helion.pl", "okno2",
    "width=800,height=400,left=0,top=0,toolbar=yes,location=yes");
```

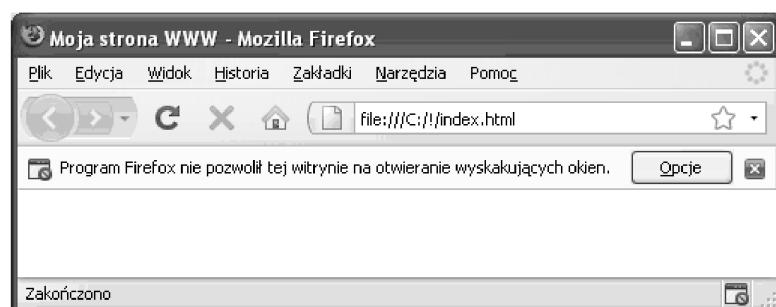
to na ekranie powinno pojawić się nowe okno o następujących parametrach:

- ◆ wczytany dokument — *http://helion.pl*,
- ◆ szerokość — 800 pikseli,
- ◆ wysokość — 400 pikseli,
- ◆ współrzędna *x* lewego górnego rogu — 0 pikseli,
- ◆ współrzędna *y* lewego górnego rogu — 0 pikseli,
- ◆ pasek narzędziowy — widoczny,
- ◆ pasek adresu — widoczny.

Jeżeli jednak w opcjach przeglądarki otwieranie nowych okien przez skrypty zostało zablokowane, albo nie zobaczymy nic, albo też pojawi się informacja o blokadzie, co można zobaczyć na rysunku 4.6.

Rysunek 4.6.

Blokada
wyskakujących okien
w przeglądarce
Firefox



Obiekt document

Obiekt `document` to reprezentacja wczytanego do przeglądarki dokumentu. Zawiera szereg właściwości, które zostały zebrane w tabeli 4.4. Część z nich istnieje tylko ze względu na kompatybilność ze starszymi wersjami przeglądarki. Należy raczej unikać korzystania z takich właściwości jak: `bgColor`, `fgColor`, `aLinkColor`, `alinkColor`, `vLink`, gdyż ich użyteczność jest znikoma. Zamiast nich stosowane są style CSS (lekcja 16.). Jeśli to tylko możliwe, należy też zrezygnować z właściwości charakterystycznych wyłącznie dla wybranych produktów, czyli `all`, `width`, `height`, bo z powodzeniem można je zastąpić przez inne konstrukcje.

Tabela 4.4. Wybrane właściwości obiektu document

Nazwa	Znaczenie	Dostępność
all	Obiekt zawierający odniesienia do wszystkich elementów dokumentu, charakterystyczny dla przeglądarki Internet Explorer.	IE, OP
alinkColor	Kolor aktywnego odnośnika.	IE, FF, OP
anchors	Tablica zawierająca odniesienia do znajdujących się w dokumencie obiektów typu Anchor.	IE, FF, OP
applets	Tablica zawierająca odniesienia do znajdujących się w dokumencie obiektów typu Applet.	IE, FF, OP
bgColor	Kolor tła dokumentu.	FF, IE
body	Obiekt zawierający treść dokumentu HTML.	IE, FF, OP
characterSet	Ciąg określający kodowanie znaków w dokumencie.	FF, OP
compatMode	Ciąg określający tryb kompatybilności dokumentu ze standardami HTML.	IE, FF, OP
cookie	Ciąg znaków zawierający cookies danego dokumentu.	IE, FF, OP
docType	Obiekt określający typ dokumentu (DTD).	FF, OP
domain	Nazwa domenowa serwera, z którego pochodzi dokument.	IE, FF, OP
embeds	Tablica zawierająca odniesienia do znajdujących się w dokumencie obiektów zagnieżdżonych.	IE, FF, OP
fgColor	Kolor tekstu dokumentu.	IE, FF
forms	Tablica zawierająca odniesienia do znajdujących się w dokumencie obiektów formularzy.	IE, FF, OP
height	Wysokość dokumentu.	FF
images	Tablica zawierająca odniesienia do znajdujących się w dokumencie obrazów.	IE, FF, OP
lastModified	Zawiera datę i czas ostatniej modyfikacji dokumentu.	IE, FF, OP
linkColor	Definiuje kolor odnośnika.	IE, FF, OP
links	Tablica zawierająca odniesienia do znajdujących się w dokumencie obiektów typu Link (odnośników).	IE, FF, OP
location	Obiekt przechowujący URL bieżącego dokumentu.	IE, FF, OP
plugins	Tablica zawierająca odniesienia do znajdujących się w dokumencie obiektów typu Plugin.	IE, FF, OP
referrer	Zawiera URL dokumentu, z którego nastąpiło odwołanie do bieżącego dokumentu.	IE, FF, OP
styleSheets	Zawiera wszystkie style zdefiniowane w dokumencie.	IE, FF, OP
title	Zawiera tytuł dokumentu zdefiniowany za pomocą znacznika <title>.	IE, FF, OP
URL	Ciąg zawierający URL bieżącego dokumentu.	IE, FF, OP
vLink	Kolor odwiedzonego odnośnika.	IE, FF, OP
width	Szerokość dokumentu.	FF

Spróbujmy wyświetlić kilka podstawowych informacji o dokumencie. Takie zadanie realizuje skrypt widoczny na listingu 4.4, a przykładowy efekt jego działania został zaprezentowany na rysunku 4.7.

Listing 4.4. Odczytanie podstawowych informacji o dokumencie

```
var str = "Podstawowe informacje o dokumencie:<br />";

str += "Tryb kompatybilności: ";
str += document.compatMode + "<br />";

str += "URL: " + document.URL + "<br />";

str += "Liczba appletów: ";
str += document.applets.length + "<br />";

str += "Liczba obrazów: ";
str += document.images.length + "<br />";

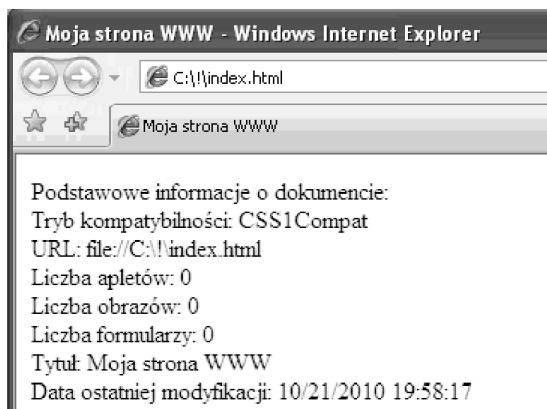
str += "Liczba formularzy: ";
str += document.forms.length + "<br />";

str += "Tytuł: " + document.title + "<br />";

str += "Data ostatniej modyfikacji: ";
str += document.lastModified + "<br />";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Rysunek 4.7.
Efekt działania skryptu odczytującego właściwości dokumentu



Dane są odczytywane z poszczególnych właściwości i dopisywane do zmiennej str, tak samo jak w innych, wcześniej prezentowanych przykładach. W przypadku liczb obrazów, odnośników oraz formularzy (ponieważ każda z właściwości, takich jak images, links, forms, może być traktowana jako tablica), dodatkowo następują odwołania do właściwości length tablic (opis tablic w rozdziale 3.).

Obiekt document nie zawiera dużej liczby metod, wybrane z nich zostały przedstawione w tabeli 4.5. Metody write i writeln poznaliśmy już w lekcji 2., w rozdziale 2. Pozostałymi zajmiemy się w kolejnej lekcji. Umożliwiają one dostęp do poszczególnych elementów witryny.

Tabela 4.5. Wybrane metody obiektu document

Metoda	Wywołanie	Opis	Dostępność
getElementById	getElementById(<i>id</i>)	Zwraca odniesienie do elementu o identyfikatorze wskazywanym przez argument <i>id</i> .	FF, IE, OP
getElementsByName ↳(nazwa)	getElementsByName ↳(nazwa)	Pobiera listę elementów posiadających atrybut name o wartości wskazanej przez argument <i>nazwa</i> .	FF, IE, OP
getElementsBy ↳TagName	getElementsByTagName ↳(tag)	Pobiera listę elementów utworzonych za pomocą znacznika określonego przez argument <i>tag</i> .	FF, IE, OP
Write	write(<i>tekst</i>)	Umieszcza w dokumencie tekst przekazany w postaci argumentu <i>tekst</i> .	FF, IE, OP
writeln	writeln(<i>tekst</i>)	Działa analogicznie do write, z tą różnicą, że na końcu tekstu przekazanego w postaci argumentu <i>tekst</i> jest dodatkowo umieszczany znak nowego wiersza.	FF, IE, OP

Obiekt history

Obiekt history przechowuje historię odwiedzin stron dokonanych przez użytkownika podczas danej sesji przeglądarki. Jest obsługiwany przez praktycznie wszystkie popularne przeglądarki. Jego właściwości (opisane w tabeli 4.6) i metody (opisane w tabeli 4.7) pozwalają na przemieszczanie się pomiędzy już odwiedzonymi stronami. Możliwość odczytu właściwości obiektu history przez skrypt może zależeć od konfiguracji przeglądarki.

Tabela 4.6. Właściwości obiektu history

Nazwa	Znaczenie	Dostępność
current	Zawiera URL aktualnego dokumentu.	OP, FF
length	Zawiera liczbę elementów przechowywanych przez obiekt history.	FF, IE, OP
next	Zawiera URL kolejnego elementu obiektu history.	FF
previous	Zawiera URL poprzedniego elementu obiektu history.	FF

Tabela 4.7. Metody obiektu history

Metoda	Opis	Dostępność
back()	Wczytuje poprzedni dokument.	FF, IE, OP
forward()	Wczytuje następny dokument.	FF, IE, OP
go(param)	Wczytuje dokument wskazywany przez argument param. Jeżeli jest to wartość całkowita, oznacza pozycję (względem bieżącego dokumentu) na liście przechowywanej przez obiekt history, jeżeli zaś parametrem jest ciąg znaków, jest on traktowany jako URL dokumentu znajdującego się na liście obiektu history.	FF, IE, OP

Obiekt location

Obiekt location reprezentuje adres URL aktualnie załadowanego dokumentu. Zawiera właściwości obrazujące składowe adresu oraz metody pozwalające na manipulację tym adresem. Właściwości zostały zebrane w tabeli 4.8. Aby lepiej zobaczyć, którym częściom adresu odpowiadają poszczególne właściwości, można uruchomić widoczny na listingu 4.5 skrypt, który wyświetla szczegółowe informacje.

Tabela 4.8. Właściwości obiektu location

Nazwa	Znaczenie	Dostępność
hash	Część adresu URL znajdująca się za znakiem #.	FF, IE, OP
host	Część adresu URL zawierająca nazwę hosta oraz numer portu.	FF, IE, OP
hostname	Część adresu URL zawierająca nazwę hosta.	FF, IE, OP
href	Pełny adres URL.	FF, IE, OP
pathname	Część adresu URL zawierająca ścieżkę dostępu i nazwę pliku.	FF, IE, OP
port	Część adresu URL zawierająca numer portu.	FF, IE, OP
protocol	Część adresu URL zawierająca nazwę protokołu (np. <i>http, ftp</i>).	FF, IE, OP
search	Część adresu URL znajdująca się za znakiem zapytania.	FF, IE, OP

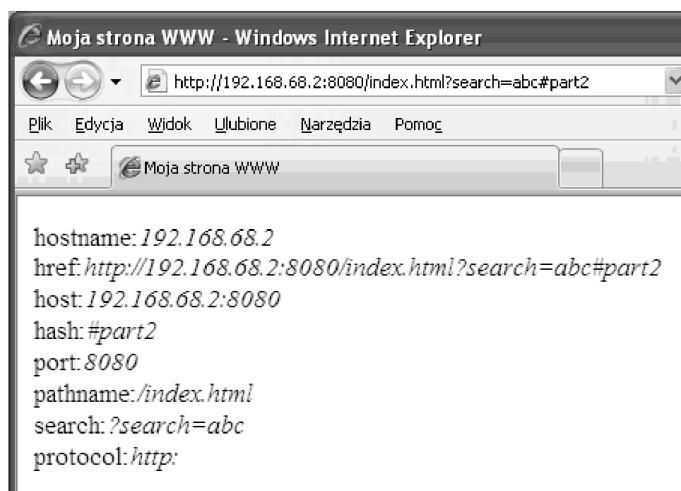
Listing 4.5. Odczyt właściwości obiektu location

```
var str = "";
for(indeks in location){
    if(typeof location[indeks] != 'function')
        str += indeks + ":<i> " + location[indeks];
        str += "</i><br />";
}
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Do odczytu właściwości obiektu `location`, a więc składowych aktualnego adresu URL, jest używana pętla `for...in`. W każdym jej przebiegu pod zmienną indeks jest podstawiana nazwa kolejnej właściwości, a więc wartość tej właściwości jest odczytywana przy użyciu konstrukcji `location[indeks]`. Ponieważ pętla odczytuje wszystkie składowe obiektu, a więc również metody (funkcje), w jej wnętrzu została umieszczona instrukcja warunkowa `if`. Za pomocą operatora `typeof` bada ona, czy typem aktualnie przetwarzanej właściwości jest `function`, a więc czy jest ona funkcją. Jeśli tak jest, odczyt jest pomijany. Dzięki temu nazwy metod nie pojawią się na stronie. Przykładowy efekt działania skryptu jest widoczny na rysunku 4.8.

Rysunek 4.8.

Odczytanie właściwości obiektu `location`



Metody obiektu `location`, których odczyt pomineliśmy w skrypcie z listingu 4.5, zostały zebrane w tabeli 4.9.

Tabela 4.9. Wybrane metody obiektu `location`

Metoda	Wywołanie	Opis	Dostępność
assign	assign(<i>url</i>)	Wczytuje dokument o adresie wskazanym przez argument <i>url</i> .	FF, IE, NN, OP
reload	reload(<i>force</i>)	Wymusza ponowne wczytanie bieżącej strony. Jeśli obecny jest argument <i>force</i> i ma on wartość true, wymusza to ponowne wczytanie zawartości witryny z serwera. W pozostałych przypadkach przeglądarka może wczytać ją z pamięci cache.	FF, IE, NN, OP
replace	replace(<i>url</i>)	Zastępuje bieżący dokument przez wczytany spod adresu wskazanego przez argument <i>url</i> . W przeciwieństwie do metody <code>assign</code> , po zastosowaniu <code>replace</code> bieżąca strona nie zostanie zapamiętana w historii przeglądanych witryn.	FF, IE, NN, OP
toString	toString()	Przekształca zawartość obiektu <code>location</code> w ciąg znaków reprezentujący przechowywany przez ten adres URL.	FF, IE, NN, OP

Warto zwrócić uwagę na różnicę między assign a replace. Obie powodują wczytanie nowej witryny, ale po zastosowaniu replace (w przeciwieństwie do assign) bieżąca strona nie zostanie zapamiętana w historii przeglądanych witryn, nie będzie więc można ponownie odwołać się do niej poprzez wcisnięcie przycisku *Wstecz* bądź też wywołanie history.go(-1) (tabela 4.7).

Metodę replace można wykorzystać np. w sytuacji, kiedy witryna została przeniesiona pod nowy adres i chcemy, aby użytkownik został automatycznie do niego przekierowany. Takie przekierowanie często jest wykonywane przez odpowiednie użycie znacznika *meta* w nagłówku strony, ale może być również wykonane za pomocą JavaScriptu. Wystarczy w skrypcie umieścić tylko jedną linię:

```
location.replace("http://nowy.adres");
```

Po zapoznaniu się z materiałem z lekcji 22. nie będziemy też mieli problemu z opóźnieniem takiego przeniesienia pod nowy adres o określony czas. Wtedy pod starym adresem można umieścić informację o nowym, wyświetlana przez zadaną liczbę sekund.

Obiekt navigator

Obiekt navigator przechowuje informacje dotyczące przeglądarki, jej nazwy, wersji, języka, systemu operacyjnego, na którym została uruchomiona itp. Jest dostępny w większości produktów i obsługuje pewien standardowy zestaw właściwości, które zostały zebrane w tabeli 4.10¹.

Przy korzystaniu z wymienionych właściwości trzeba wziąć pod uwagę, że przeglądarki dosyć swobodnie podchodzą do wypełniania ich danymi. Nie ma tu jasno określonego standardu, do którego dopasowywałyby się wszystkie produkty. Wystarczy skorzystać z pętli for...in w postaci analogicznej do przedstawionej na listingu 4.5, ale operującej na obiekcie navigator. Będzie wtedy miała postać:

```
for(indeks in navigator){  
    if(typeof navigator[indeks] != 'function')  
        str += indeks + ":<i> " + navigator[indeks];  
        str += "</i><br />";  
}
```

Jeśli wstawimy ją do kodu z listingu 4.5 i tak powstały skrypt uruchomimy w kilku przeglądarkach, bardzo szybko odkryjemy różnice występujące między nimi. Na rysunkach 4.9 i 4.10 zostały przedstawione efekty działania takiego skryptu w przeglądarkach Firefox 3 i Internet Explorer 7. Widać wyraźnie, że przedstawiane przez przeglądarki informacje mogą wprowadzać w błąd.

¹ Spotyka się również kilka innych właściwości specyficznych dla konkretnych przeglądarek. Niektóre nie są nawet opisane w dokumentacjach technicznych udostępnianych przez producentów.

Tabela 4.10. Wybrane właściwości obiektu navigator

Nazwa	Znaczenie	Dostępność
appCodeName	Nazwa kodowa przeglądarki.	FF, IE, OP
appName	Oficjalna nazwa przeglądarki.	FF, IE, OP
appVersion	Wersja kodowa przeglądarki.	FF, IE, OP
appMinorVersion	Podwersja przeglądarki.	IE, OP
cookieEnabled	Określa, czy w przeglądarce jest włączona obsługa cookies (true — tak, false — nie).	FF, IE, OP
cpuClass	Rodzina procesorów urządzenia, na którym jest uruchomiona przeglądarka.	IE
language	Kod języka przeglądarki.	FF, OP
mimetypes	Tablica zawierająca listę typów MIME obsługiwanych przez przeglądarkę. W niektórych przypadkach właściwość ta jest wartością pustą.	FF, IE, OP
online	Określa, czy przeglądarka pracuje w trybie online.	FF, IE
oscpu	Określa system operacyjny, na którym jest uruchomiona przeglądarka.	FF
platform	Określa platformę systemową, dla której jest przeznaczona przeglądarka.	FF, IE, OP
plugins	Tablica zawierająca odniesienia do rozszerzeń zainstalowanych w przeglądarce.	FF, IE, OP
product	Nazwa produktowa przeglądarki (np. Gecko).	FF
productSub	Wersja produktowa przeglądarki.	FF
systemLanguage	Język systemu operacyjnego, na którym jest uruchomiona przeglądarka.	IE
userAgent	Ciąg wysyłany przez przeglądarkę do serwera jako nagłówek <code>HTTP_USER_AGENT</code> . Z reguły pozwala na dokładną identyfikację przeglądarki.	FF, IE, OP
userLanguage	Język użytkownika (z reguły wersja językowa przeglądarki).	IE, OP
vendor	Dostawca (producent) przeglądarki.	FF
vendorSub	Numer produktu według producenta (np. 5.1, 6.0).	FF

Rysunek 4.9.

Właściwości obiektu navigator w przeglądarce Firefox



Rysunek 4.10.

*Właściwości
obiektu navigator
w przeglądarce
Internet Explorer*



Najdokładniejsze informacje uzyskamy, odwołując się do właściwości userAgent, w której najczęściej można odnaleźć wszystkie niezbędne informacje i to nawet wtedy, kiedy identyfikacja produktu została zmieniona w jego opcjach konfiguracyjnych (część przeglądarek udostępnia taką możliwość)². Oto kilka przykładowych opisów, które można tam znaleźć³:

Mozilla/5.0 (Windows; U; Windows NT 6.0; pl; rv:1.9.0.3) Gecko/2008092417 Firefox/3.0.3
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 3.5.21022; FDM)
Opera/9.51 (X11; Linux i686; U; Fedora; en)

W pierwszej chwili nie wygląda to zbyt czytelnie, ale bliższa analiza pozwoli się zorientować, że:

- ◆ w pierwszym przypadku mamy do czynienia z przeglądarką Firefox w wersji 3.0.3 pracującą pod kontrolą systemu Windows Vista (NT 6.0);
- ◆ w drugim przypadku mamy do czynienia z przeglądarką Internet Explorer w wersji 6.0 (MSIE 6.0) pracującą pod kontrolą systemu Windows XP (NT 5.1);
- ◆ w trzecim przypadku mamy do czynienia z przeglądarką Opera w wersji 9.51 pracującą pod kontrolą systemu Linux (dystrybucja Fedora).

W oparciu o te spostrzeżenia można napisać prosty skrypt rozpoznający ze sporym prawdopodobieństwem typ przeglądarki, z którą mamy do czynienia. Wystarczy sprawdzić, czy w ciągu znajdującym się we właściwości userAgent znajdują się ciągi firefox, msie i opera. Takie zadanie realizuje kod widoczny na listingu 4.6.

Listing 4.6. Rozpoznanie typu przeglądarki

```
var agent = navigator.userAgent.toLowerCase();
var str = "Zakładam, że twoja przeglądarka to ":

if(agent.indexOf('firefox') != -1){
    str += "Firefox.";
}
}
```

² Niestety, w niewielu przypadkach spotkamy się także z sytuacją, kiedy przeglądarka nie udostępnia informacji we właściwości userAgent (może się tam znajdować np. pusty串 znaków).

³ Pod adresem <http://www.user-agents.org/> znajduje się internetowa baza danych, w której można znaleźć wiele innych przykładów.

```
else if(agent.indexOf('opera') != -1){  
    str += "Opera.:";  
}  
else if(agent.indexOf('msie') != -1){  
    str += "Internet Explorer.:";  
}  
else{  
    str = "Nie mogę rozpoznać typu przeglądarki.:";  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Tekst znajdujący się we właściwości userAgent jest konwertowany za pomocą funkcji toLowerCase (opis obiektów typu String w rozdziale 5., w lekcji 17.), dzięki czemu będzie zawierał wyłącznie małe litery. To ułatwia dalsze operacje porównywania. Po konwersji tekst jest zapisywany w pomocniczej zmiennej agent. Następnie za pomocą złożonej instrukcji warunkowej i metody indexOf jest sprawdzane, czy w ciągu zapisanym w agent znajduje się któryś ze słów charakterystycznych dla jednej z rozpoznawanych przeglądarek, czyli firefox, opera i msie. Jeśli tak, do zmiennej str jest dopisywana nazwa rozpoznanego produktu. Kiedy żadna z przeglądarek nie zostanie rozpoznana, zmiennej str przypisywany jest odpowiedni komunikat. Użyta metoda indexOf zwraca indeks wystąpienia poszukiwanego ciągu znaków lub też wartość -1, jeśli dany ciąg nie został odnaleziony. Operacjami na ciągach znaków zajmiemy się bliżej dopiero w lekcji 17.

Przedstawiony tu sposób rozpoznania przeglądarki jest bardzo uproszczony. W praktyce może być konieczne zastosowanie dużo bardziej złożonej analizy. Nie trzeba jednak wykonywać jej samodzielnie. W internecie można znaleźć wiele gotowych i darmowych skryptów bardzo dobrze wykonujących takie zadanie. Badanie, z którym z produktów mamy do czynienia, jest potrzebne, bo — jak już dało się zauważyć — przeglądarki nie są w pełni kompatybilne ze sobą. To często powoduje konieczność pisania różnych wersji kodu skryptu.

Lekcja 13. DOM — dostęp do elementów witryny

Witryna WWW w przeglądarce jest interpretowana jako struktura hierarchiczna, a poszczególne jej elementy budowane ze znaczników (X)HTML — jako obiekty posiadające swoje właściwości i metody. Stosując JavaScript, można uzyskać dostęp do tych obiektów i wpływać na ich postać, wygląd oraz zachowanie. Można również tworzyć dynamicznie całkiem nowe. Tak więc JavaScript sprawia, że mamy praktycznie pełną kontrolę nad witryną i jej zawartością. W lekcji 13. dowiemy się, jak uzyskać dostęp do konkretnego elementu witryny, jak nim manipulować, jak tworzyć nowe elementy, a także usuwać już istniejące.

Struktura dokumentu

Po wczytaniu dokumentu (X)HTML przeglądarka interpretuje go i tworzy własną reprezentację pozwalającą jej na wyświetlenie strony i wykonywanie związków z tym zadań (np. reakcję na działanie użytkownika). W tej reprezentacji dokument (X)HTML jest strukturą hierarchiczną, zgodną z modelem DOM (ang. *Document Object Model*), i tworzy drzewo, którego węzłami są elementy tego dokumentu. Jak takie drzewo wygląda, możemy podejrzeć np. w przeglądarce Firefox, która zawiera rozszerzenie o nazwie *Inspektor DOM* (ang. *DOM Inspector*). W wersjach do 3. rozszerzenie to jest dostępne standardowo (o ile zostanie wybrane w opcjach instalacyjnych pakietu), w wersjach od 3.0 należy zainstalować je jako rozszerzenie⁴.

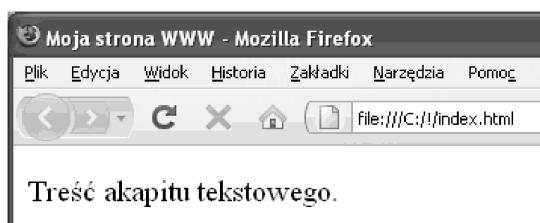
Wczytajmy do tej przeglądarki prosty dokument o treści widocznej na listingu 4.7.

Listing 4.7. Dokument HTML zawierający akapit tekstowy

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
</head>
<body>
<div id="dataDiv">
<p id="pt1">Treść akapitu tekstuowego.</p>
</div>
</body>
</html>
```

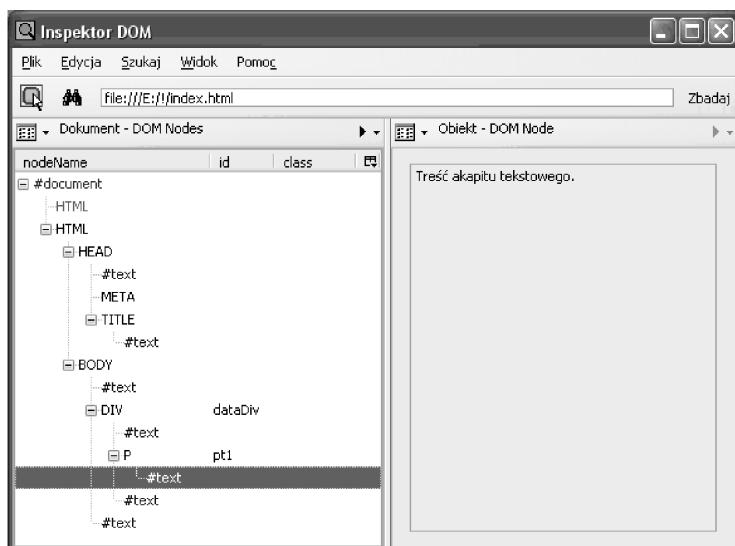
Oczywiście, na stronie będzie widoczna jedynie treść akapitu tekstuowego, taka jak na rysunku 4.11. Jeśli jednak wywołamy *Inspektora DOM*, co można zrobić, wybierając z menu *Narzędzia* pozycję *Inspektor DOM* lub wciskając kombinację klawiszy *Ctrl+Shift+I*, zobaczymy, że dokument faktycznie jest hierarchiczną strukturą (drzewem), której węzłami są poszczególne elementy witryny (rysunek 4.12).

Rysunek 4.11.
Wygląd witryny
z listingu 4.7



⁴ Należy je odszukać na stronie <https://addons.mozilla.org/>. Powinno być dostępne pod adresem <https://addons.mozilla.org/en-US/firefox/addon/6622>.

Rysunek 4.12.
Struktura witryny
w Inspektorze DOM



Jak widać na rysunku 4.12, element (węzeł) HEAD zawiera w sobie węzły META i TITLE, a element BODY — węzeł DIV, który z kolei zawiera węzeł P. Oprócz wymienionych węzłów zwykłych (ang. *node*) związanych ze znacznikami HTML, widoczne są również węzły tekstowe (ang. *text node*, na rysunku 4.12 oznaczone jako `#tekst`), które zawierają tekstową zawartość danego węzła zwykłego. W powyższym przykładzie węzeł P zawiera węzeł tekstowy z ciągiem znaków Treść akapitu tekstu (treść zawartą w węzłach tekstowych można obserwować w module z prawej strony inspektora). Zawartość węzłów tekstowych to tekst zapisany w znacznikach (X)HTML.

Docieklewy programista spyta na pewno, skąd w takim razie wzięło się tak wiele węzłów tekstowych widocznych na rysunku 4.12? Nie dość, że nie widać żadnej ich zawartości, to wydaje się również, że w kodzie z listingu 2.1 nie ma odpowiadającej im treści. Czy aby na pewno? Przyjrzymy się dokładniej. Pierwszym podwęzłem węzła BODY jest `#text` (czyli nasz „niby” pusty węzeł tekstowy), a kolejnym DIV (rysunek 4.12). Odpowiadający temu fragmentowi kod HTML ma natomiast postać:

```
<body>
<div id="dataDiv">
```

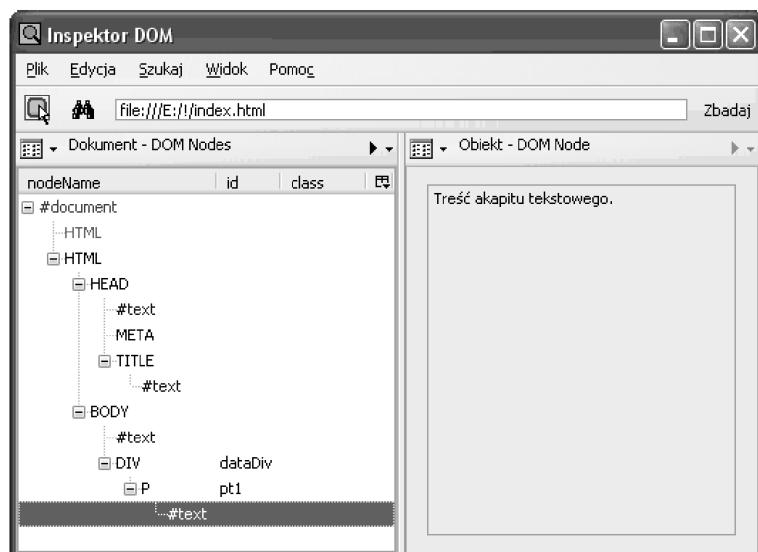
Czy zatem przeglądarka zinterpretowała dane prawidłowo? Oczywiście, choć zależy to od sposobu interpretacji zapisu. Otóż, choć w pierwszej chwili może się wydawać, że między znacznikiem `<body>` a `<div>` nie ma nic, to znajduje się tam przecież znak końca wiersza. Skoro tak, Firefox tworzy odpowiadający mu (a widoczny na rysunku 4.12) węzeł tekstowy. Możemy się o tym przekonać w prosty sposób. Zmieńmy dotychczasowy kod na widoczny na listingu 4.7, tak aby cała sekcja `<body>` przyjęła postać z listingu 4.8.

Listing 4.8. Usunięcie części znaków końca wiersza

```
<body>
<div id="dataDiv"><p id="pt1">Treść akapitu tekstu.</p></div>
```

Fragment strony został tu zapisany jednym ciągiem. Jest przez to mniej czytelny dla człowieka, ale przecież przeglądarki nie sprawia to żadnej różnicy, ponieważ podczas interpretacji kodu i tak pomija stosowane do formatowania znaki końca wiersza. Sprawdźmy więc, jak wygląda taka strona w *Inspektorze DOM* (rysunek 4.13). Widać teraz wyraźnie, że część węzłów tekstowych znikła.

Rysunek 4.13.
Część węzłów
tekstowych
została usunięta



W typowych zastosowaniach istnienie bądź nieistnienie tych nadmiarowych węzłów tekstowych nie ma praktycznego znaczenia. Dostęp do elementów strony najczęściej bowiem uzyskujemy za pomocą metody `getElementById` i właściwości `innerHTML`, czym zajmiemy się już w kolejnej części lekcji. Gdybyśmy jednak chcieli poruszać się bezpośrednio po drzewie DOM dokumentu (co czasem jest niezbędne), dodatkowe węzły tekstowe muszą być brane pod uwagę.

Dostęp do elementów strony WWW

Za pomocą JavaScriptu można manipulować elementami strony WWW, zmieniać ich treść, style itp. Aby jednak można było to zrobić, trzeba najpierw uzyskać dostęp do danego elementu. Najczęściej używamy w tym celu metody `getElementById` obiektu `document` (tabela 4.5 w lekcji 12.). Pobiera ona obiekt zdefiniowany za pomocą znacznika (X)HTML posiadającego atrybut `id` o wartości przekazanej jako argument `getElementById`. Jeśli zatem w sekcji `<body>` znajdzie się warstwa zdefiniowana jako:

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

to aby pobrać odwołanie do niej, należy użyć instrukcji:

```
document.getElementById("warstwaDanych");
```

Pobrany obiekt najlepiej przypisać zmiennej:

```
var div = document.getElementById("warstwaDanych");
```

Wtedy zmienną `div` (można, oczywiście, zmienić jej nazwę) możemy traktować jak obiekt reprezentujący warstwę o identyfikatorze warstwy danych (obiekt ten jest węzłem drzewa DOM).

Po uzyskaniu dostępu do danego elementu strony możemy nim manipulować, np. zmienić zawartość. Do tego celu bardzo często używa się właściwości `innerHTML`. Otóż, większość węzłów drzewa DOM powiązanych ze znacznikami HTML ma taką właściwość i zawiera ona kod HTML występujący w danym elemencie. Ujmując rzecz nieco dokładniej, należałoby napisać, że `innerHTML` to zawartość danego elementu strony w postaci kodu HTML, taka, jaką „widzi” przeglądarka. Zawartość ta może być i odczytywana, i zapisywana, a więc istnieje możliwość zarówno sprawdzenia, jaką treść zawiera dany element (znacznik), jak i zmiany tej zawartości.

Zatem odczyt kodu HTML naszej hipotetycznej warstwy warstwy danych można przeprowadzić za pomocą instrukcji:

```
var tekstHTML = div.innerHTML;
```

a zapis przy użyciu:

```
div.innerHTML = "zawartość warstwy";
```

To, co zapiszemy we właściwości `innerHTML`, zostanie potraktowane jak zawartość danego elementu (znacznika).

Jasne jest już na pewno, dlaczego stosowane do tej pory przykłady miały w sekcji `<body>` zdefiniowaną warstwę `dataDiv`:

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

a w kodach skryptów występował fragment:

```
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Pierwsza instrukcja pobierała bowiem odwołanie do warstwy o identyfikatorze `dataDiv`, a druga zmieniała jej treść na zawartość zmiennej `str`. O tym, że tak jest w istocie, przekonamy się za pomocą *Inspektora DOM*. Umieścmy w sekcji `<body>` pliku `index.html` lub `index.xhtml` treść z listingu 4.9, a w pliku `skrypt.js` skrypt widoczny na listingu 4.10.

Listing 4.9. Definicja sekcji `<body>`

```
<body>
  <div id="dataDiv">
    <p id="pt1">Treść akapitu tekstowego.</p>
  </div>
  <script type="text/javascript" src="skrypt.js">
  </script>
  <noscript>
    <p>Twoja przeglądarka nie obsługuje skryptów.</p>
  </noscript>
</body>
```

Listing 4.10. Zmiana danych za pomocą właściwości innerHTML

```
var str = "<p>Zamieniona treść akapitu tekstowego.</p>";  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

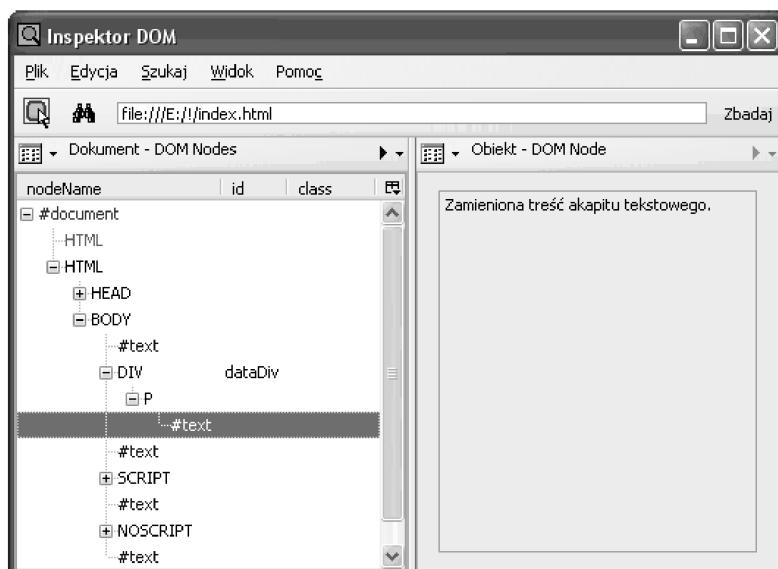
Po wczytaniu przykładu do przeglądarki nie będziemy w stanie zobaczyć pierwotnej treści akapitu zdefiniowanej w kodzie HTML, zostanie ona bowiem natychmiast zmieniona przez skrypt. Dokładniej, cała treść warstwy dataDiv ulegnie wymianie na treść zmiennej str, a więc akapit zostanie ponownie zdefiniowany jako:

```
<p>Zamieniona treść akapitu tekstowego.</p>
```

Jeśli uruchomimy *Inspektora DOM* (rysunek 4.14), przekonamy się, że tak jest w istocie. Warstwa dataDiv zawiera jeden akapit tekstowy (węzeł P, zwróciły uwagę, że nie ma on identyfikatora id, a akapit kodzie HTML miał go), a treścią tego akapitu jest ciąg znaków Zamieniona treść akapitu tekstowego.

Rysunek 4.14.

Inspektor DOM
ułatwia zmienioną
treść witryny

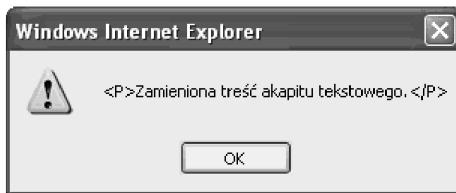


Przeprowadźmy jeszcze jeden eksperyment i dodajmy na końcu kodu skryptu z listingu 4.9 instrukcję:

```
alert(dataDiv.innerHTML);
```

Wyświetli ona w oknie dialogowym kod warstwy dataDiv. Uruchommy tak zmieniony skrypt w różnych przeglądarkach. Jaki będzie efekt? W Firefoxie tekst znacznika będzie taki sam jak zdefiniowany w kodzie skryptu, ale już w Internet Explorerze czy Operze będzie nieco inaczej. Spójrzmy na rysunek 4.15. Jak widać, zmienił się znacznik <p>. W kodzie pisany był małą literą, a wyświetlany jest wielką. Nawet więc na tak prostym przykładzie widać, że przeglądarka może nieco inaczej widzieć kod dokumentu.

Rysunek 4.15.
Odczyt zawartości właściwości innerHTML

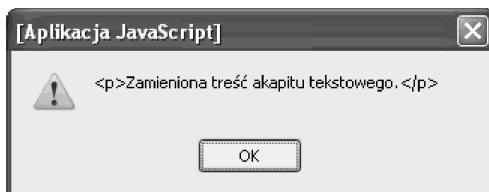


To, oczywiście, nie dotyczy tylko Internet Explorera i Opery. Jeśli znacznik `<p>` w kodzie zapiszemy wielkimi literami:

```
var str = "<P>Zamieniona treść akapitu tekstowego.</P>";
```

Firefox zaprezentuje tę treść małymi, co jest widoczne na rysunku 4.16.

Rysunek 4.16.
Również Firefox dokonał korekty znacznika



Bezpośrednia manipulacja węzłami dokumentu

W poprzednim punkcie poznaliśmy właściwość `innerHTML`, która pozwala w bardzo wygodny sposób manipulować zawartością elementów witryny. Jej zastosowanie jest bardzo proste, bowiem nie wymaga bezpośredniego tworzenia i modyfikacji węzłów drzewa dokumentu, a całą pracę związaną z przetwarzaniem przypisywanego tej właściwości kodu wykonuje przeglądarka. Niestety, `innerHTML` nie jest częścią standardu HTML (choć nie można wykluczyć, że w przyszłości nią się stanie), czasami też chcielibyśmy mieć możliwość pełnej kontroli nad węzłami drzewa DOM, a tym samym, bezpośredniej edycji elementów dokumentu. To również jest możliwe, choć zwykle wymaga od programisty dużo więcej pracy. Daje też jednak większą kontrolę nad zawartością i strukturą dokumentu.

Spójrzmy więc ponownie na rysunek 4.12. Wynika z niego, że jeśli w kodzie HTML pojawi się fragment:

```
<div id="warstwaDanych">
<p id="pt1">Treść akapitu tekstowego.</p>
</div>
```

to po zinterpretowaniu przez przeglądarkę spowoduje on powstanie węzła (ang. *node*) DIV, zawierającego jeden zwykły węzeł potomny P, który z kolei będzie zawierał jeden tekstowy węzeł (ang. *text node*) potomny. Treścią węzła tekstu będzie napis: Warstwa zawierająca tekst.

Jeżeli zatem chcemy bezpośrednio manipulować węzłami, musimy dowiedzieć się dwóch rzeczy. Po pierwsze — jak dostać się do węzła, po drugie — jak odczytać lub zmodyfikować jego zawartość. Odpowiedzią na pierwsze pytanie jest właściwość

childNodes. Zawiera ona odwołania do wszystkich węzłów potomnych danego węzła (w uproszczeniu można ją potraktować jak tablicę obiektów). Operacje na zawartości wybranego węzła można natomiast wykonywać, odwołując się do jego właściwości nodeValue. Jeśli więc za pomocą metody getElementById pobierzemy odwołanie do jakiegoś elementu strony, np. warstwy dataDiv, i zapiszemy je w zmiennej element:

```
var element = document.getElementById("dataDiv");
```

lista wszystkich bezpośrednich węzłów potomnych będzie dostępna przez właściwość childNodes:

```
element.childNodes
```

I tak dostęp do pierwszego (o indeksie 0!) węzła osiągniemy za pomocą odwołania:

```
element.childNodes[0]
```

Jeżeli natomiast będziemy chcieli odczytać wartość tego węzła i zapisać ją w zmiennej value, użyjemy instrukcji:

```
var value = element.childNodes[0].nodeValue;
```

Spróbujmy więc zmienić treść znajdująca się na witrynie, manipulując bezpośrednio węzłami dokumentu. Przykład takiej operacji został zaprezentowany na listingu 4.11. Skrypt ten współpracuje z kodem HTML z listingu 4.9.

Listing 4.11. Bezpośrednia zmiana zawartości elementu strony

```
var str = "Zamieniona treść akapitu tekstuowego.";
var pt1 = document.getElementById("pt1");
var pt1TextNode = pt1.childNodes[0];
pt1TextNode.nodeValue = str;
```

Zasada działania tego skryptu jest bardzo prosta. Skoro w kodzie HTML został zdefiniowany akapit tekstowy o atrybutie id równym pt1, odwołanie do niego można pobrać za pomocą instrukcji:

```
var pt1 = document.getElementById("pt1");
```

Ten akapit zawiera pierwotny tekst: Treść akapitu tekstuowego, który jest zdefiniowany w pierwszym (i jedynym) węźle potomnym węzła pt1 (rysunek 4.12). Ów węzeł potomny jest węzłem tekstem. Odwołanie do niego otrzymujemy przy użyciu instrukcji:

```
var pt1TextNode = pt1.childNodes[0];
```

Aby zmienić tekst widniejący na stronie (zdefiniowany w akapicie), trzeba zmienić wartość właściwości nodeValue węzła (obiektu) pt1TextNode. Robimy to za pomocą instrukcji:

```
pt1TextNode.nodeValue = str;
```

A zatem modyfikacja treści strony przez bezpośrednią manipulację wartościami węzłów dokumentu HTML, choć bardziej złożona niż używanie właściwości innerHTML, wcale nie jest skomplikowana.

Tworzenie elementów strony przez skrypt

Możliwości JavaScriptu i DOM nie kończą się, oczywiście, tylko na modyfikacji istniejących węzłów dokumentu. Nic nie stoi na przeszkodzie, aby do istniejącej struktury dodawać nowe elementy. W takim wypadku niezbędne jest jednak rozróżnianie typów węzłów. Otóż, węzły zwykłe, związane ze znacznikami, np.: <p>, <div>, , tworzy się za pomocą metody createElement obiektu document — argumentem powinna być nazwa znacznika. Oto przykład:

```
document.createElement("div")
```

W ten sposób buduje się nowe elementy witryny. Węzły tekstowe (czyli tekstową zawartość znaczników) tworzy się natomiast, stosując metodę createTextNode — argumentem powinien być tekst, który ma być umieszczony w węźle, np.:

```
document.createTextNode("Tekst zawarty na warstwie")
```

Węzły łączymy ze sobą, korzystając z metody appendChild. Jeśli zatem wykonamy instrukcję:

```
var divEl = document.createElement("div");
var textEl = document.createTextNode("tekst");
```

zmienna divEl będzie zawierała węzeł definiujący nową warstwę, natomiast textEl — nowy węzeł tekstowy. Aby dołączyć węzeł tekstowy do warstwy, tak by stał się jej węzłem potomnym, należy użyć metody appendChild:

```
divEl.appendChild(textEl);
```

Oczywiście, żeby tak zdefiniowana warstwa wraz z tekstem znalazła się w dokumencie, należy dołączyć ją w którymś jego miejscu, również za pomocą metody appendChild. Zobaczmy, jak to będzie wyglądało w praktyce. Z pliku *index.html* z listingu 4.9 usunijmy akapit tekstowy <p>, pozostawiając pustą warstwę dataDiv:

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

A w kodzie skryptu umieścmy treść z listingu 4.12.

Listing 4.12. Tworzenie elementów dokumentu HTML

```
var str = "Zamieniona treść akapitu tekstowego.";

var div = document.getElementById("dataDiv");
var pEl = document.createElement("p");
var pElTextNode = document.createTextNode(str);

pEl.appendChild(pElTextNode);
div.appendChild(pEl);
```

Jak widać, dynamiczne tworzenie nowych elementów dokumentu również nie jest skomplikowane. Najpierw pobierane jest odwołanie do warstwy dataDiv — to do niej zostanie dodany nowy węzeł. Następnie tworzony jest nowy element dokumentu, czyli akapit tekstowy definiowany za pomocą znacznika <p>:

```
var pEl = document.createElement("p");
```

Aby tekst mógł się znaleźć w akapicie, musi on zawierać tekstowy węzeł potomny. Dlatego też taki element jest tworzony za pomocą metody createTextNode:

```
var pElTextNode = document.createTextNode(str);
```

Jako jej argument została użyta zawartość zmiennej str zawierającej przykładowy napis.

Tym samym, po wykonaniu wymienionych instrukcji:

- ◆ zmienna div zawiera wskazanie do warstwy warstwaDanych,
- ◆ zmienna pEl — nowy akapit tekstowy,
- ◆ zmienna pElTextNode — węzeł tekstowy z przykładowym tekstem.

Pozostaje połączyć powstałe elementy w całość. Węzeł pElTextNode jest więc dodawany do węzła pEl:

```
pEl.appendChild(pElTextNode);
```

A węzeł pEl do węzła div:

```
div.appendChild(pEl);
```

Dzięki temu dokument HTML zostaje uzupełniony o dodatkowy akapit tekstowy, którego treść pojawi się na stronie.

Usuwanie elementów strony

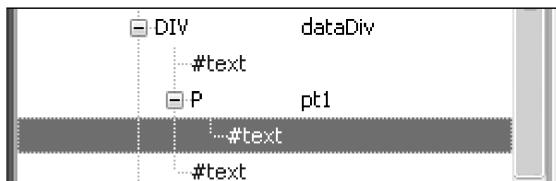
Skoro elementy dokumentu mogą być tworzone i dodawane dynamicznie, musi też istnieć możliwość ich usuwania. Służy do tego metoda removeChild. Zastosowana w stosunku do obiektu (elementu HTML, węzła drzewa DOM) powoduje usunięcie węzła potomnego przekazanego w postaci argumentu. Jej wywołanie ma schematyczną postać:

```
element.removeChild(obj);
```

gdzie *element* to element, z którego ma być usuwany węzeł potomny, a *obj* to usuwany element (węzeł potomny). Rezultatem działania tej metody jest wskazanie (referencja) do usuniętego elementu. Należy zwrócić uwagę, że w tym przypadku usunięcie oznacza faktycznie odłączenie węzła potomnego (*obj*) od nadrzędnego (*element*), a nie fizyczne usunięcie go z pamięci. Po tej operacji odłączany węzeł nadal istnieje i może być ponownie użyty.

Powróćmy więc ponownie do kodu HTML z listingu 4.9. W warstwie dataDiv znalazły się tam akapit tekstowy oraz dwa węzły tekstowe odpowiadające znakom końca linii (co jest widoczne na rysunku 4.17). Postarajmy się więc programowo, za pomocą skryptu, usunąć całą zawartość tej warstwy. Można to zrobić przy użyciu skryptu wiadocznego na listingu 4.13.

Rysunek 4.17.
Zawartość warstwy
dataDiv w postaci
drzewa DOM



Listing 4.13. Programowe usunięcie elementów dokumentu

```

var dataDiv = document.getElementById("dataDiv");
var liczbaWęzłów = dataDiv.childNodes.length;
for(i = liczbaWęzłów - 1; i >= 0; i--){
    dataDiv.removeChild(dataDiv.childNodes[i]);
}
  
```

Odrońanie do warstwy dataDiv jest pobierane w tradycyjny sposób, za pomocą metody getElementById. Następnie pobierana jest liczba węzłów potomnych. Jest to wartość właściwości length tablicy childNodes zawierającej wszystkie elementy potomne warstwy. Węzły potomne są usuwane w pętli for z wykorzystaniem metody removeChild. Argumentem każdego wywołania tej metody jest kolejny węzeł do usunięcia, który uzyskujemy, stosując odwołanie dataDiv.childNodes[i].

Gdy uruchomimy skrypt, na stronie nie pojawią się żadne dane, to znak, że faktycznie udało nam się programowo usunąć zawartość warstwy dataDiv. Jeśli dodatkowo uruchomimy *Inspektora DOM*, zobaczymy, że faktycznie węzeł DIV nie ma żadnych węzłów potomnych (rysunek 4.18). Cała operacja zakończyła się więc sukcesem.

Rysunek 4.18.
Węzeł DIV stracił
wszystkie węzły
potomne



Zwróćmy przy tym uwagę, że zmienna iteracyjna pętli for zmienia się w zakresie od liczbaWęzłów - 1 do 0, czyli usuwamy węzły od ostatniego (childNodes[liczbaWęzłów - 1]) do pierwszego (childNodes[0]). Dlaczego tak? A co by się stało, gdyby pętla miała postać inkrementacyjną:

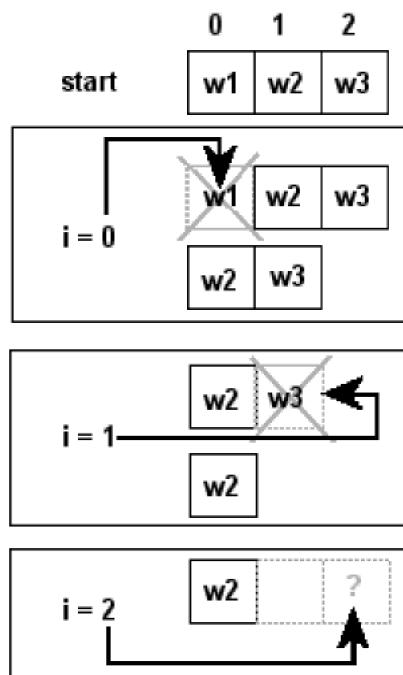
```

for(i = 0; i < liczbaWęzłów; i++){
    dataDiv.removeChild(dataDiv.childNodes[i]);
}
  
```

Załóżmy istnienie trzech węzłów o indeksach (0, 1 i 2) — zmienna i zmieniałaby się w zakresie od 0 do 2. Poszczególne etapy usuwania danych z tablicy childNodes w takiej sytuacji można zobaczyć na rysunku 4.19. W pierwszym przebiegu i miałoby wartość 0 i zostały usunięty węzeł o indeksie 0. Tym samym, pozostałyby dwa węzły, ale ich indeksy uległy zmianie. Dotychczasowy drugi (o indeksie 1) stałby się pierwszym (o indeksie 0), a dotychczasowy trzeci (o indeksie 2) — drugim (o indeksie 1). W drugim przebiegu i miałoby wartość 1 i zostały usunięty węzeł o indeksie 1 (a więc ten, który pierwotnie był trzeci!). Pozostał jeden węzeł (ten, który pierwotnie był drugi), a jego indeks zmieniłby się na 0. W trzecim przebiegu pętli i miałoby wartość 2, a w tablicy childNodes znajdowałby się tylko jeden węzeł o indeksie 0. Próba usunięcia zakończyłaby się zgłoszeniem wyjątku i zatrzymaniem kodu skryptu.

Rysunek 4.19.

Schemat nieprawidłowego usuwania elementów



Przekonajmy się teraz, że metoda removeChild faktycznie jedynie odłącza wskazany węzeł potomny od węzła nadziędnego, ale nie niszczy go i nie usuwa z pamięci. Napiszemy skrypt, który, przełączając węzły, przeniesie akapit tekstowy wraz z całą zawartością z jednej warstwy do drugiej. Najpierw utworzymy kod HTML. Został on zaprezentowany na listingu 4.14.

Listing 4.14. Kod HTML strony z dwoma warstwami i stylami CSS

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
<style type="text/css">
```

```
.dd1, .dd2{  
    width : 200px;  
    height : 30px;  
}  
.dd1{  
    background-color : #f0f0f0;  
}  
.dd2{  
    background-color : #c0c0c0;  
}  
</style>  
</head>  
<body>  
<div id="dataDiv1" class="dd1">  
    <p id="pt1">Treść akapitu tekstowego.</p>  
</div>  
<div id="dataDiv2" class="dd2">  
</div>  
<script type="text/javascript" src="skrypt.js">  
</script>  
<noscript>  
    <p>Twoja przeglądarka nie obsługuje skryptów.</p>  
</noscript>  
</body>  
</html>
```

W sekcji `<head>` został umieszczony znacznik `<style>` definiujący style CSS dla warstw. Dzięki temu będą wyraźnie rozróżnione na witrynie, tak byśmy mogli łatwo zorientować się w ich położeniu i zawartości. Zostały zdefiniowane klasy dd1 i dd2. Atrybutami wspólnymi są `width` i `height` określające szerokość i wysokość. Każda warstwa będzie zatem miała 100 pikseli szerokości i 20 pikseli wysokości. Osobno zostały natomiast zdefiniowane kolory tła (atribut `background-color`). Dla stylu dd1 jest to `#f0f0f0`, a dla dd2 — `#c0c0c0` (są to odcienie szarości).

W sekcji body umieszczone zostały dwie warstwy: dataDiv1 i dataDiv2. Pierwsza otrzymała styl dd1, a druga — dd2. Łatwo więc rozróżnić je na stronie. W warstwie dataDiv1 znalazła się także akapit tekstowy zdefiniowany za pomocą znacznika `<p>`. Akapit ten zawiera przykładowy tekst. Naszym zadaniem jest odłączenie akapitu od warstwy dataDiv1 i dodanie go do warstwy dataDiv2. Musimy zatem napisać skrypt, którego efekt działania będzie taki, jak na rysunku 4.20. Widać na nim, że pierwszy blok (warstwa dataDiv1) jest pusty, a tekst znajduje się w bloku drugim (warstwie dataDiv2), czyli dokładnie odwrotnie, niż zostało to zdefiniowane w kodzie HTML. Skrypt, który wykona takie zadanie, jest widoczny na listingu 4.15.

Na początek pobieramy odwołania do obu znajdujących się w kodzie HTML warstw, dataDiv1 i dataDiv2, oraz do akapitu pt1. W tym celu używamy standardowej metody `getElementById`. Następnie wykonujemy dwie proste instrukcje. Najpierw odłączamy cały akapit od warstwy dataDiv1 — w tym celu używamy metody `removeChild`:

```
dataDiv1.removeChild(pt1);
```

Rysunek 4.20.

Akapit został przemieszczony do drugiej warstwy

**Listing 4.15. Przelaczanie węzłów DOM**

```
var dataDiv1 = document.getElementById("dataDiv1");
var dataDiv2 = document.getElementById("dataDiv2");

var pt1 = document.getElementById("pt1");

dataDiv1.removeChild(pt1);
dataDiv2.appendChild(pt1);
```

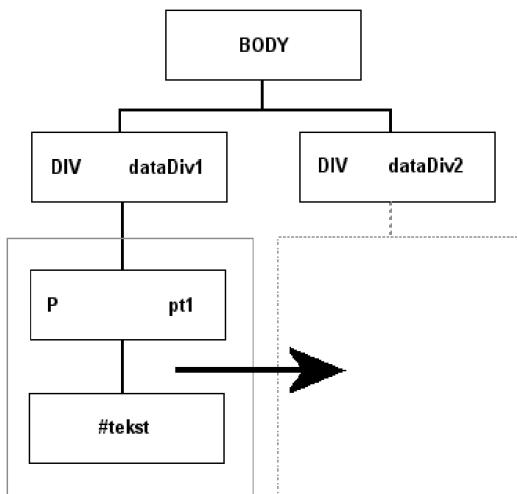
Później dodajemy akapit do warstwy dataDiv2 — w tym celu używamy metody appendChild:

```
dataDiv2.appendChild(pt1);
```

Schematycznie operacje te można przedstawić tak, jak na rysunku 4.21. Można powiedzieć, że przełączliśmy akapit z jednej warstwy do drugiej.

Rysunek 4.21.

Schemat przełączania węzłów



Ćwiczenia do samodzielnego wykonania

Ćwiczenie 13.1.

Zmodyfikuj kod z listingu 4.10, tak aby nie była tracona pierwotna zawartość akapitu tekstowego. Wykonaj dwa warianty ćwiczenia. W pierwszym na warstwie mają się pojawić oba akapity. W drugim — do pierwszego akapitu (zdefiniowanego w kodzie (X)HTML) ma zostać dodana treść drugiego.

Ćwiczenie 13.2.

Umieść w kodzie (X)HTML warstwę zdefiniowaną za pomocą znacznika `<div>`. Napisz skrypt, który doda do tej warstwy elementy, jakie w postaci HTML miałyby postać: `<i>Praktyczny kurs</i>`. Nie używaj właściwości `innerHTML`.

Ćwiczenie 13.3.

Zmień kod z listingu 4.13 tak, aby zmienna iteracyjna i pętli `for` była zwiększana (począwszy od 0), a nie zmniejszana, a skrypt działał prawidłowo.

Lekcja 14. Zdarzenia

Lekcja 14. jest poświęcona zdarzeniom. Termin zdarzenie (ang. *event*) należy rozumieć zgodnie z intuicją. Może to być przesunięcie kurSORA myszy, kliknięcie, załadowanie strony, opuszczenie strony itp. Reagowaniem na zdarzenia zajmują się tzw. procedury obsługi zdarzeń (ang. *event handlers*). To funkcje JavaScript wywoływanie w reakcji na dane zdarzenie. Sprawdzimy więc, jakie występują typy zdarzeń, jak wiązać zdarzenia z procedurami obsługi i jakie występują przy tym różnice między przeglądarkami. Nauczymy się reagować na kliknięcia i ruchy myszy. Umiejętność obsługi zdarzeń bardzo przyda się w kolejnych lekcjach.

Obsługa zdarzeń

Po co reagować na zdarzenia? Wydaje się to dosyć oczywiste. Chcemy, aby nasze witryny były jak najbardziej atrakcyjne i interaktywne, zatem musimy reagować na działania użytkownika. A jest to możliwe tylko dzięki obsłudze zdarzeń. Przy czym zdarzenie to zarówno to, co zachodzi w świecie realnym, czyli kliknięcie przez użytkownika przycisku, jak i powiązany z tym pewien bitt wirtualny. Kiedy coś dzieje się na stronie WWW, przeglądarka generuje *zdarzenie (event)*. Powstaje wtedy obiekt zdarzenia, który je opisuje, oraz badane jest, czy istnieje procedura obsługi. Jeśli tak, jest wykonywana.

Niestety, w różnych przeglądarkach zastosowano odmienne modele obsługi zdarzeń. W lekcji skupimy się na modelu podstawowym, historycznie najstarszym, obsługiwany podobnie przez wszystkie dostępne produkty. Oprócz modelu podstawowego, istnieje jeszcze nowocześniejszy, ustandaryzowany model DOM (w kolejnych wersjach: level 1, level 2 i level 3), a także model specyficzny dla przeglądarek z rodziny Internet Explorer. Model podstawowy, na którym się skupimy, nazywany jest też czasem modelem DOM level 0.

Istnieją różne typy zdarzeń, powstające w odpowiedzi na konkretne działania na stronie WWW. Innego typu zdarzenie powstaje w przypadku kliknięcia, a inne — w wyniku załadowania strony. W modelu DOM level 0 zazwyczaj zdarzenie utożsamiamy z nazwą procedury obsługi (*event handler*), która jest jednocześnie nazwą atrybutu znacznika (X)HTML tworzącego element strony. Schematycznie wygląda to tak:

```
<znacznik onzdarzenie="instrukcje;">
```

Formalnie znaczy to, że dla zdarzenia o typie *zdarzenie* procedurą obsługi jest *onzdarzenie* i procedurze tej przypisano instrukcję JavaScriptu *instrukcje*. Gdy np. element strony zostanie kliknięty, powstaje zdarzenie typu *click*, a jego procedura obsługi ma nazwę *onclick*. Jeśli więc chcemy, aby w wyniku kliknięcia tego elementu, np. warstwy, została wykonana instrukcja `alert('abc');`, napiszemy:

```
<div onclick="alert('abc');">
```

W takim przypisaniu może znaleźć się też kilka instrukcji:

```
<znacznik onzdarzenie="instrukcja1: instrukcja2: instrukcja3;">
```

Ponieważ jednak kod procedury obsługi zazwyczaj składa się z większej liczby instrukcji i wykonuje bardziej zaawansowane zadania, często w znaczniku (X)HTML przypisuje się jedynie wywołanie funkcji:

```
<znacznik onzdarzenie="nazwa_funkcji();">
```

Wtedy kod funkcji obsługującej zdarzenie umieszczamy w skrypcie JavaScript. Głównie tę właśnie formę będziemy stosować w dalszej części książki. Jeśli skrypt umieszczany jest bezpośrednio w kodzie (X)HTML, całość ma schematyczną strukturę:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
<script type="text/javascript">
    function procedura_obslugi()
    {
        //kod funkcji obsługującej zdarzenie
    }
    //pozostała część skryptu
</script>
</head>
```

```
<body>
<znacznik onzdarzenie="procedura_obsługi();">
    <!-- treść znacznika -->
</znacznik>
</body>
</html>
```

W dalszej części książki będziemy stosować nieco uproszczoną terminologię. Otóż, to, co jest de facto nazwą procedury obsługi, będziemy nazywać zdarzeniem, a napisaną przez nas funkcję wywoływaną w odpowiedzi na zdarzenie — procedurą obsługi zdarzenia. Jeśli więc w kodzie (X)HTML pojawi się konstrukcja w schematycznej postaci:

```
<znacznik onzdarzenie="nazwa_funkcji();">
```

powiemy, że zdarzeniu onzdarzenie elementu `<znacznik>` została przypisana procedura obsługi w postaci funkcji nazwa_funkcji, np. dla kodu:

```
<div id="dataDiv" onclick="dataDivOnClick();">
```

powiemy, że zdarzeniu onclick warstwy dataDiv została przypisana procedura obsługi w postaci funkcji dataDivOnClick.

Lista typowych zdarzeń, wraz z elementami strony WWW, których mogą dotyczyć, została przedstawiona w tabeli 4.11.

Tabela 4.11. Lista typowych zdarzeń

Nazwa zdarzenia	Opis	Elementy HTML obsługujące zdarzenie
onabort	Zdarzenie, które powstaje, kiedy zostanie przerwane ładowanie obrazu.	img
onblur	Zdarzenie, które powstaje, kiedy element traci fokus.	większość elementów strony
onchange	Zdarzenie, które powstaje, kiedy element traci fokus i jednocześnie zmienia się wartość zawarta w tym elemencie (np. polu tekstowym).	input, select, textarea
onclick	Zdarzenie, które powstaje, kiedy element został kliknięty.	większość elementów strony
ondblclick	Zdarzenie, które powstaje, kiedy element został dwukrotnie kliknięty.	większość elementów strony
onerror	Zdarzenie, które powstaje, kiedy w trakcie ładowania obrazu wystąpi błąd.	img
onfocus	Zdarzenie, które powstaje, kiedy element otrzymuje fokus.	większość elementów strony
onkeydown	Zdarzenie, które powstaje, kiedy naciśnięty zostanie klawisz klawiatury.	większość elementów strony
onkeypress	Zdarzenie, które powstaje, kiedy klawisz klawiatury zostanie naciśnięty i puszczyony.	większość elementów strony
onkeyup	Zdarzenie, które powstaje, kiedy klawisz klawiatury zostanie puszczyony.	większość elementów strony

Tabela 4.11. Lista typowych zdarzeń (ciąg dalszy)

Nazwa zdarzenia	Opis	Elementy HTML obsługujące zdarzenie
onload	Zdarzenie, które powstaje, kiedy przeglądarka zakończy ładowanie strony lub ramki.	body, frameset
onmousedown	Zdarzenie, które powstaje, kiedy klawisz myszy zostanie naciśnięty nad elementem.	większość elementów strony
onmousemove	Zdarzenie, które powstaje, kiedy kurSOR myszy jest przesuwany nad elementem.	większość elementów strony
onmouseout	Zdarzenie, które powstaje, kiedy kurSOR myszy opuści obszar elementu.	większość elementów strony
onmouseover	Zdarzenie, które powstaje, kiedy kurSOR myszy wejdzie w obszar elementu.	większość elementów strony
onmouseup	Zdarzenie, które powstaje, kiedy klawisz myszy zostanie zwolniony nad elementem.	większość elementów strony
onreset	Zdarzenie, które powstaje podczas resetowania (wywołanie w kodzie metody <code>reset</code> , kliknięcie przycisku <code>reset</code>) formularza.	form
onresize	Zdarzenie, które powstaje, gdy zmieni się rozmiar okna.	body, frameset
onselect	Zdarzenie, które powstaje podczas zaznaczania fragmentu tekstu.	input (text, textarea)
onsubmit	Zdarzenie, które powstaje podczas wysyłania (wywołanie w kodzie metody <code>submit</code> , kliknięcie przycisku <code>submit</code>) formularza.	form
onunload	Zdarzenie, które powstaje, kiedy przeglądarka usuwa bieżący dokument.	body, frameset

Ładowanie strony

Jednym ze zdarzeń wymienionych w tabeli 4.11 jest `onload`. Powstaje ono, gdy dokument (X)HTML zostanie załadowany do przeglądarki. Użyjemy go do sprawdzenia, jak w praktyce wygląda obsługa zdarzeń. Na początku wykonajmy prosty test. Spowodujemy, że po załadowaniu dokumentu na ekranie pojawi się okno dialogowe. W taki właśnie sposób działa kod widoczny na listingu 4.16. Jest to kod HTML. W wersji XHTML kod, a także działanie sekcji `<body>` będą identyczne (zmienią się jedynie nagłówki i deklaracja dokumentu).

Listing 4.16. Wykorzystanie zdarzenia `onload`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
</head>
```

```
<body onload="alert('Strona została załadowana.');//>
<div>
    To jest treść witryny.
</div>
</body>
</html>
```

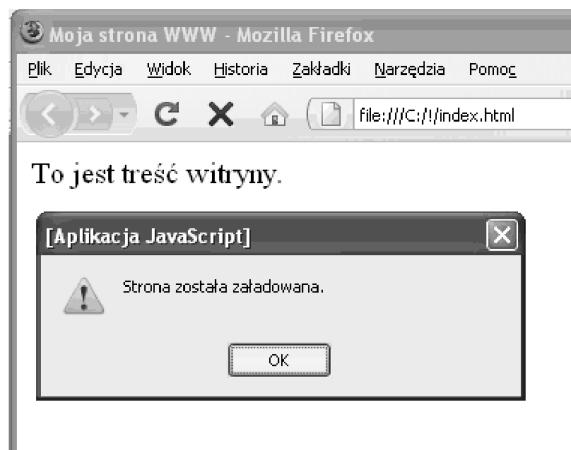
Jest to typowa strona HTML, zawierająca w sekcji <body> warstwę zdefiniowaną za pomocą znacznika <div>. Zawartość sekcji <body> w tej chwili nie jest istotna. Interesuje nas przede wszystkim sama postać znacznika <body>:

```
<body onload="alert('Strona została załadowana.');//>
```

Zawiera on atrybut onload. Oznacza to, że zdarzeniu onload⁵ została przypisana procedura obsługi. W tym przypadku ową procedurą jest po prostu wywołanie funkcji (metody) alert z prostym komunikatem tekstowym. A zatem po wczytaniu strony do przeglądarki powstanie zdarzenie onload. Ponieważ została mu przypisana procedura obsługi, zostanie wykonana. To spowoduje, że na ekranie pojawi się okno dialogowe, jakie można zobaczyć na rysunku 4.22.

Rysunek 4.22.

Po wczytaniu strony na ekranie pojawiło się okno dialogowe



Podobny przykład wykonywaliśmy już w lekcji 1. Tam też na ekranie było wyświetlane okno dialogowe. Zwróćmy jednak uwagę na różnice. Otoż, jeśli w skrypcie w sekcji <head> umieściliśmy instrukcje JavaScriptu, były one wykonywane, zanim strona została przetworzona. Jeśli kod znalazł się w sekcji <body>, najpierw była wyświetlana część strony znajdująca się przed skryptem, potem wykonywany skrypt, a następnie przetwarzana część strony znajdująca się za skryptem. W przypadku użycia zdarzenia onload jest inaczej. Mamy gwarancję, że przypisany mu kod JavaScript zostanie wykonany po całkowitym załadowaniu (przetworzeniu) witryny i przygotowaniu jej do wyświetlenia. To bardzo ważna cecha. Dzięki temu możemy w procedurze obsługi onload umieścić kod operujący na dowolnych elementach strony, mając pewność, że są one gotowe (a nie trwa ich przetwarzanie przez przeglądarkę).

⁵ Formalnie — zdarzeniu load, ale jak zostało to wspomniane wcześniej, stosujemy uproszczoną terminologię.

Jeżeli zamiast bezpośredniego przypisania kodu do znacznika chcemy skorzystać z funkcji (któրą będziemy nazywać procedurą obsługi danego zdarzenia), powinniśmy zastosować kod widoczny na listingu 4.17 (podobnie jak w listingu 4.16, wersja dla XHTML będzie miała taką samą postać z dokładnością do nagłówków i deklaracji typu dokumentu).

Listing 4.17. Funkcja jako procedura obsługi zdarzenia

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
<script type="text/javascript">
    function onLoadHandler()
    {
        alert("Strona została załadowana.");
    }
</script>
</head>
<body onload="onLoadHandler();">
<div>
    To jest treść witryny.
</div>
</body>
</html>
```

Efekt działania tego kodu będzie taki sam jak kodu z listingu 4.16. Różnica dotyczy sposobu obsługi zdarzenia. Tym razem w odpowiedzi na zdarzenie wywoływana jest funkcja o nazwie `onLoadHandler`, czyli zdarzeniu `onload` została przypisana procedura obsługi w postaci tej funkcji:

```
<body onload="onLoadHandler();">
```

Definicja tej funkcji znajduje się w skrypcie umieszczonym w sekcji `<head>`. Jednym zadaniem wykonywanym przez `onLoadHandler` jest wywołanie metody `alert` powodującej wyświetlenie okna dialogowego. Warto zwrócić uwagę na nazewnictwo funkcji będących procedurami obsługi zdarzeń. W tym przypadku `onLoadHandler` oznacza procedurę obsługi (hadler) zdarzenia `onload`. Poszczególne człony nazwy zostały wyróżnione wielkimi literami, dzięki czemu jest czytelniejsza. Ponieważ zdarzenie `onload` dotyczy zawsze całej strony i występuje tylko raz, taka nazwa będzie bardzo dobra. Oczywiście, można stosować dowolne nazwy funkcji, ale lepiej, jeśli jednoznacznie wskazują, jakiego zdarzenia i jakiego elementu dotyczą. Przykładowo zamiast `onLoadHandler` można zastosować nazwy: `bodyOnLoad`, `onLoadBody`, `bodyLoad`. Jeśli zdarzenie dotyczy jakiegoś elementu witryny, warto użyć nazwy tego elementu i nazwy zdarzenia, np. dla zdarzenia `onclick` i warstwy o identyfikatorze `dataDiv` dobrymi nazwami byłyby `dataDivClick`, `dataDivOnClick`, `onClickDataDiv` itp. Konsekwentne stosowanie odpowiedniego schematu nazw zwiększa czytelność kodu i ułatwia jego analizę.

Oprócz zdarzenia `onload`, istnieje także `onunload`. Jak łatwo się domyślić, powstaje ono, gdy strona jest usuwana z przeglądarki — gdy użytkownik ją opuszcza (np. przechodząc na inną witrynę)⁶. Można je wykorzystać do uruchomienia kodu wykonującego czynności „sprzątające”. Zdarzenie `onunload` jest rzadko używane, nam jednak posłuży do pokazania, że znacznikowi można przypisać wiele zdarzeń, oczywiście tylko takich, które dany element strony obsługuje. Schematycznie taka konstrukcja wygląda następująco:

```
<znacznik onzdarzenie1="instrukcje1;"  
          onzdarzenie2="instrukcje2;"  
          onzdarzenie3="instrukcje3;"  
          <!-- itd. -->  
>
```

Dla znacznika `<body>` oraz zdarzeń `onload` i `onunload` moglibyśmy zastosować kod:

```
<body onload="alert('Strona została załadowana.');"   
       onunload="alert('Strona jest opuszczana.');"   
>
```

Gdyby natomiast obsługą zdarzeń miały zająć się osobne funkcje (procedury obsługi), kod:

```
<body onload="onLoadHandler();"  
       onunload="onUnloadHandler();"  
>
```

W tym drugim przypadku, aby osiągnąć zamierzony efekt, należałoby — oczywiście — umieścić w treści strony skrypt zawierający definicje funkcji `onLoadHandler` i `onUnloadHandler`, np. taki, jaki został zaprezentowany na listingu 4.18.

Listing 4.18. Procedury obsługi dla zdarzeń `onload` i `onunload`

```
<script type="text/javascript">  
    function onLoadHandler()  
    {  
        alert("Strona została załadowana.");  
    }  
    function onUnloadHandler()  
    {  
        alert("Strona jest opuszczana.");  
    }  
</script>
```

⁶ Uwaga: przeglądarka Opera obsługuje zdarzenie `onunload` dopiero od wersji 9., a robi to w sposób niestandardowy. W tej przeglądarce nie każda zmiana strony powoduje obsługę zdarzenia.

Reagowanie na kliknięcia

Jednym z częściej wykorzystywanych zdarzeń jest `onclick` (`click`). Powstaje, gdy zostanie naciśnięty przycisk myszy, czyli zostanie kliknięty jakiś element strony⁷. Skoro tak, pozwala w bezpośredni sposób reagować na poczynania użytkownika witryny. Procedurę obsługi zdarzenia `onlick` można przypisać większości elementów strony. Sprawdźmy to na przykładzie prostej warstwy generowanej za pomocą znacznika `<div>`. W sekcji `<body>` witryny wystarczy umieścić kod widoczny na listingu 4.19.

Listing 4.19. Warstwa reagująca na kliknięcia

```
<body>
    <div id="dataDiv"
        style="background-color:silver;width:200px;height:20px;
               text-align:center;" 
        onclick="alert('To działa!');"
    >
        To jest treść warstwy.
    </div>
</body>
```

Warstwa otrzymała identyfikator `dataDiv` oraz został jej przypisany styl wyróżniający ją na stronie: szerokość — 200 pikseli, wysokość — 20 pikseli, kolor tła — szary (srebrny), tekst — wyśrodkowany w poziomie. Została jej także przypisana procedura obsługi zdarzania `onclick`, która jest instrukcją wywołującą metodę `alert`. Będzie ona wywoływana po każdym kliknięciu warstwy, a zatem każde kliknięcie spowoduje pojawienie się okna dialogowego.

Teraz przyjrzyjmy się nieco bardziej złożonemu przykładowi. Jeśli wykorzystamy informacje z lekcji 13., możemy napisać skrypt, który po kliknięciu warstwy zmieni jej zawartość. W taki właśnie sposób działa kod z listingu 4.20.

Listing 4.20. Zmiana zawartości warstwy po jej kliknięciu

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Moja strona WWW</title>
    <script type="text/javascript">
        function dataDivClick()
        {
            var dataDiv = document.getElementById("dataDiv");
            dataDiv.innerHTML = "Warstwa została kliknięta.";
        }
    </script>
</head>
```

⁷ Podobnym zdarzeniem jest `ondblclick` powstające, gdy element witryny zostanie dwukrotnie kliknięty. Obsługa `click` i `dblclick` jest identyczna. Dalsze przykłady będą dotyczyć zdarzenia `click`.

```
<body>
  <div id="dataDiv"
    style="background-color:silver;width:200px;height:20px;
           text-align:center;" 
    onclick="dataDivClick();"
  >
    To jest treść warstwy.
  </div>
</body>
</html>
```

Warstwa została zdefiniowana w podobny sposób jak na listingu 4.19. Różnica jest taka, że tym razem zdarzeniu `onclick` została przypisana procedura obsługi w postaci funkcji `dataDivClick`:

```
onclick="dataDivClick();"
```

Funkcja została umieszczona w skrypcie znajdującym się w sekcji `<head>` i wykonuje dwie bardzo proste czynności. Najpierw za pomocą metody `getElementById` obiektu `document` pobiera odwołanie do warstwy `dataDiv`:

```
var dataDiv = document.getElementById("dataDiv");
```

Następnie właściwości `innerHTML` obiektu `dataDiv` przypisuje przykładowy tekst:

```
dataDiv.innerHTML = "Warstwa została kliknięta.";
```

Tym samym, po kliknięciu warstwy tekst Warstwa została kliknięta. stanie się jej treścią i pojawi się na ekranie.

Użyjmy teraz omawianego zdarzenia w bardziej praktycznym przykładzie. Postaramy się zamienić zwykłe akapity tekstowe, definiowane za pomocą znacznika `<p>`, na odnośniki. A więc kliknięcie akapitu będzie powodowało przeniesienie użytkownika pod wskazany adres WWW. Działający w ten sposób kod został zaprezentowany na listingu 4.21.

Listing 4.21. Akapity tekstowe udające odnośniki

```
<body>
  <p style="cursor:pointer;" 
     onclick="location.href='http://helion.pl'">
    Witryna helion.pl
  </p>
  <p style="cursor:pointer;" 
     onclick="location.href='http://marcinlis.com'">
    Witryna marcinlis.com
  </p>
  <p style="cursor:pointer;" 
     onclick="location.href='http://onepress.pl'">
    Witryna onepress.pl
  </p>
</body>
```

Na listingu uwzględniona została jedynie treść sekcji <body>, pozostałe elementy strony są bowiem standardowe i nie wymagają omawiania. W kodzie znalazły się trzy akapity. Każdy z nich otrzymał styl:

```
"
```

dzięki czemu po najechaniu kursorem na tekst obraz kurSORA zmieni się na wskaźnik, tak jak w przypadku zwykłego odnośnika zdefiniowanego za pomocą znacznika <a>. Ważniejszy jednak jest drugi atrybut znacznika <p>:

```
onclick="location.href='http://adres.www'"
```

Przypisuje on zdarzeniu onclick instrukcję JavaScript:

```
location.href='http://adres.www'
```

Jak wiemy z lekcji 12., powoduje ona zmianę bieżącej strony na wskazaną przez adres http://adres.www. Tak więc kliknięcie danego akapitu spowoduje wczytanie strony znajdującej się pod przypisanym mu adresem URL.

Reagowanie na ruchy myszy

Gdy zajrzymy do tabeli 4.11, znajdziemy zdarzenia o nazwach onmouseover i onmouseout. Pozwalają one na kontrolę ruchów myszy, a dokładniej — na wykrycie, kiedy kurSOR pojawi się w obrębie danego elementu (nad danym elementem — zdarzenie onmouseover), oraz kiedy obszar tego elementu opuszcza (zdarzenie onmouseout). Sprawdźmy, jak to działa. Na listingu 4.22 jest widoczna treść przykładowej sekcji <body>.

Listing 4.22. Reakcja na zdarzenie onmouseover

```
<body>
  <div id="dataDiv"
    style="background-color:silver;width:200px;height:20px;
           text-align:center;"
    onmouseover="alert('KurSOR jest nad warstwą.')";
  >
    To jest treść warstwy.
  </div>
</body>
```

W treści strony za pomocą znacznika <div> została zdefiniowana warstwa. Jest podobna do tej z listingu 4.19, jednak zamiast zdarzenia onclick zostało użyte onmouseover:

```
onmouseover="alert('KurSOR jest nad warstwą.');"
```

To oznacza, że w momencie gdy kurSOR myszy znajdzie się nad warstwą, zostanie wykonana instrukcja:

```
alert('KurSOR jest nad warstwą.');
```

A tym samym, będzie wywołana metoda alert. To z kolei spowoduje pojawienie się na ekranie okna dialogowego.

Po uruchomieniu kodu przekonamy się, że zdarzenie onmouseover faktycznie funkcjonuje w opisany sposób. Użyjmy go zatem w połączeniu z onmouseout. W poprzedniej części lekcji powstał kod, w którym akapity tekstowe udawały odnośniki. Spróbujmy tak zmienić jego działanie, aby po najechaniu na dany akapit nie tylko zmieniał się rodzaj kurSORA, ale też żeby zamiast treści akapitu pojawiła się przypisany mu adres URL w czystej postaci. Po usunięciu kurSORA znad akapitu jego treść powinna powracać do pierwotnej postaci.

Pracę zaczniemy od napisania funkcji JavaScript, która będzie się zajmowała wymianą tekstu w wybranym elemencie strony. Przyjmie ona postać widoczną na listingu 4.23, którego treść należy umieścić w sekcji <head> witryny.

Listing 4.23. Funkcja wymieniająca treść elementu witryny

```
<script type="text/javascript">
    function zmieńTekst(id, tekst)
    {
        var el = document.getElementById(id);
        el.innerHTML = tekst;
    }
</script>
```

Funkcja zmieńTekst przyjmuje dwa argumenty: id i tekst. Pierwszy określa identyfikator elementu strony, którego treść ma zostać wymieniona, a drugi — nową zawartość elementu (w tym przypadku tekst akapitu). Funkcja jest uniwersalna i można jej użyć do wymiany treści praktycznie dowolnego elementu. Najpierw pobiera ona odwołanie do elementu posiadającego atrybut id o wartości wskazanej przez argument id:

```
var el = document.getElementById(id);
```

A następnie właściwości innerHTML tego elementu przypisuje wartość argumentu tekst:

```
el.innerHTML = tekst;
```

Funkcji tej użyjemy w sposób widoczny na listingu 4.24.

Listing 4.24. Zmiana treści akapitów tekstowych

```
<body>
<p style="cursor:pointer;" id="pt1"
    onclick="location.href='http://helion.pl'"
    onmouseover="zmieńTekst('pt1', 'http://helion.pl')"
    onmouseout="zmieńTekst('pt1', 'Witryna helion.pl')"
>
    Witryna helion.pl
</p>
<p style="cursor:pointer;" id="pt2"
    onclick="location.href='http://marcinlis.com'"
    onmouseover="zmieńTekst('pt2', 'http://marcinlis.com')"
    onmouseout="zmieńTekst('pt2', 'Witryna marcinlis.com')"
>
    Witryna marcinlis.com
```

```
</p>
<p style="cursor:pointer;" id="pt3"
    onclick="location.href='http://onepress.pl'"
    onmouseover="zmieńTekst('pt3', 'http://onepress.pl')"
    onmouseout="zmieńTekst('pt3', 'Witryna onepress.pl')"
>
Witryna onepress.pl
</p>
</body>
```

Ogólna struktura witryny pozostała taka sama jak w przykładzie z listingu 4.21, choć zmieniły się definicje akapitów. Każdy z nich otrzymał atrybut `id`, dzięki któremu można je jednoznacznie zidentyfikować. Oprócz atrybutu `id`, pojawiły się także `onmouseover` i `onmouseout`. Mają one schematyczną postać:

```
onmouseover="zmieńTekst('id_akapitu', 'odnośnik')"
onmouseout="zmieńTekst('id_akapitu', 'tekst_akapitu')"
```

A zatem zdarzeniu `onmouseover` przypisane zostało wywołanie funkcji `zmieńTekst`, w którym pierwszy argument odpowiada identyfikatorowi akapitu, a drugi — odnośnikowi przypisanemu temu akapitowi. Zdarzeniu `onmouseout` zostało natomiast przypisane wywołanie tej samej funkcji, której jednak jako drugi argument został przekazany tekst przypisany akapitowi. To oznacza, że po najechaniu na dany akapit kursorem myszy jego treść zostanie zamieniona na odpowiadający akapitowi adres URL, a po usunięciu kurSORA nad akapitem zostanie przywrócona oryginalna treść.

Dynamiczne przypisywanie procedur obsługi

W dotychczasowych przykładach przypisywanie zdarzeniom procedur odbywało się statycznie. Takie powiązania dokonywane były w kodzie (X)HTML i nie zmieniały się w trakcie działania witryny. Jeśli zatem jakiś element strony był definiowany w ogólnej postaci:

```
<znacznik onzdarzenie="procedura_obsługi();">
```

to `procedura_obsługi` była funkcją JavaScript zajmującą się obsługą zdarzenia, która nie zmieniała się w trakcie działania strony.

Nic jednak nie stoi na przeszkodzie, aby wiązanie zdarzenia z jego procedurą obsługi odbywało się dynamicznie. Elementy (X)HTML mają bowiem właściwości o nazwach zgodnych z nazwami zdarzeń, które pozwalają na dokonanie takiego przypisania. Jeśli zatem np. za pomocą metody `getElementById` uzyskamy odwołanie do jakiegoś elementu witryny i zapiszemy je w zmiennej `obj`:

```
var obj = document.getElementById("id_elementu");
```

to procedurę obsługi zdarzenia przypiszemy przy użyciu instrukcji:

```
obj.onzdarzenie = procedura_obsługi();
```

W przypadku zdarzenia onclick (click) będzie to wyglądało tak:

```
obj.onclick = procedura_obsługi();
```

Wykonajmy ilustrujący to przykład. Umieścimy na stronie dwie warstwy. Pierwszej, statycznie, w kodzie (X)HTML przypiszemy procedurę obsługi zdarzenia onclick. Druga początkowo nie będzie obsługiwała żadnych zdarzeń. Dopiero kliknięcie pierwszej warstwy spowoduje przypisanie procedury obsługi zdarzeniu onclick drugiej warstwy. Wtedy też zacznie ona reagować na kliknięcia. Zaczniemy od kodu (X)HTML. Został przedstawiony na listingu 4.25 (jest to treść sekcji <body> witryny, jednakowa dla kodu HTML i XHTML).

Listing 4.25. Warstwy z obsługą kliknięć

```
<body>
    <div id="div1"
        style="background-color:#e0e0e0; width:200px; height:20px;
               text-align:center;"
        onclick="div1Click();"
    >
        To jest pierwsza warstwa.
    </div>
    <div id="div2"
        style="background-color:#d0d0d0; width:200px; height:20px;
               text-align:center;"
    >
        To jest druga warstwa.
    </div>
</body>
```

W sekcji <body> znajdują się dwie umieszczone jedna pod drugą warstwy. Mają przypisane style pozwalające wyróżnić je na ekranie oraz identyfikatory div1 i div2. Każda zawiera również przykładowy tekst. Pierwszej warstwie (div1) została przypisana procedura obsługi zdarzenia onclick w postaci funkcji div1Click — będzie więc reagowała na kliknięcia. Drugiej warstwie (div2) takiej procedury nie przypisano, zatem kliknięcie jej tuż po wczytaniu kodu (X)HTML do przeglądarki nie spowoduje żadnej reakcji.

Czas przygotować treść funkcji div1Click. Została przedstawiona na listingu 4.26. Treść tego skryptu należy umieścić w sekcji <head>.

Listing 4.26. Funkcja przypisująca procedurę obsługi zdarzenia

```
<script type="text/javascript">
    function div1Click()
    {
        var div2 = document.getElementById('div2');
        div2.onclick = function(){
            alert('Druga warstwa została kliknięta.');
        }
        alert('Druga warstwa będzie reagowała na kliknięcia.');
    }
</script>
```

Zadaniem funkcji `div1Click` jest przypisanie procedury obsługi zdarzeniu `onclick` warstwy `div2`, tak by zaczęła ona reagować na kliknięcie. Najpierw za pomocą metody `getElementById` jest więc pobierane (i zapisywane w zmiennej `div2`) odwołanie do warstwy `div2`. Następnie właściwości `onclick` obiektu `div2` jest przypisywana funkcja anonimowa. To ona będzie procedurą obsługi zdarzenia `onclick`, a więc będzie wywoływana po każdym kliknięciu warstwy. Funkcja ta zawiera jedynie wywołanie metody `alert` wyświetlającej okno dialogowe. Po dokonaniu przypisania jest wyświetlna informacja o tym, że druga warstwa będzie reagowała na kliknięcia:

```
alert('Druga warstwa będzie reagowała na kliknięcia.');
```

Gdy wczytamy kod do przeglądarki i klikniemy drugą warstwę, nic się nie stanie. Jeżeli jednak klikniemy pierwszą, drugiej zostanie przypisana procedura obsługi zdarzenia `onclick`. Wtedy też druga warstwa zacznie reagować na kliknięcia, wyświetlając stosowny komunikat.

Oczywiście, dynamicznie przypisywaną procedurą obsługi nie musi być funkcja anonimowa. Z równie dobrym skutkiem możemy użyć zwykłej funkcji. Wystarczy, jeśli zamiast kodu z listingu 4.26 użyjemy tego z listingu 4.27. Efekt końcowy będzie identyczny.

Listing 4.27. *Zwykła funkcja jako dynamiczna procedura obsługi*

```
<script type="text/javascript">
    function div2Click()
    {
        alert('Druga warstwa została kliknięta.');
    }
    function div1Click()
    {
        var div2 = document.getElementById('div2');
        div2.onclick = div2Click;
        alert('Druga warstwa będzie reagowała na kliknięcia.');
    }
</script>
```

Zwróćmy jednak uwagę na pewien mankament występujący w kodzie z dwóch ostatnich listingów. Otóż, każde kliknięcie warstwy `div1` powoduje wywołanie funkcji `div1Click`. Ponieważ funkcja ta przypisuje procedurę obsługi zdarzenia `onclick` warstwy `div2`, to czynność ta jest wykonywana wielokrotnie, a nie ma przecież takiej potrzeby. Nie jest to błąd, który w jakikolwiek sposób zaburzałby działanie witryny, ale nie jest to najlepsze rozwiązanie. Po co bowiem wielokrotnie przypisywać tę samą procedurę, skoro można to zrobić tylko raz, po pierwszym kliknięciu warstwy `div1`. Rozwiążanie tego problemu pozostawiamy jednak jako ćwiczenie do samodzielnego wykonania (ćwiczenie 14.2).

Zdarzenia i dynamiczne elementy dokumentu

W przykładach z poprzedniej części lekcji pokazaliśmy, że procedury obsługi zdarzeń mogą być przypisywane elementom strony dynamicznie, w trakcie jej działania. Jeśli połączymy tę informację z materiałem przedstawionym w lekcji 13., okaże się, że procedury obsługi zdarzeń można bez problemów przypisywać dynamicznie tworzonym elementem witryny. To bardzo dobra wiadomość, gdybyśmy bowiem dynamicznie tworzyli elementy witryny, a nie było możliwości dynamicznego przypisywania im procedur obsługi zdarzeń, obiekty te nie mogłyby odpowiednio reagować na działania użytkownika. Na szczęście, przy obsłudze zdarzeń nie ma znaczenia, w jaki sposób został utworzony dany element. O tym, że tak jest naprawdę, powinien przekonać kod przedstawiony na listingu 4.28.

Listing 4.28. Procedury obsługi dla dynamicznych elementów strony

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Moja strona WWW</title>
<script type="text/javascript">
    function createDiv()
    {
        var divEl = document.createElement("div");
        divEl.onclick = function(){
            alert("Warstwa została kliknięta.");
        }

        var str = "To jest przykładowa warstwa.";
        var divElTextNode = document.createTextNode(str);

        divEl.appendChild(divElTextNode);
        document.body.appendChild(divEl);
    }
</script>
</head>
<body onload="createDiv();">
    <!-- tutaj zostanie dynamicznie wstawiona warstwa -->
</body>
</html>
```

Tym razem treść sekcji <body> jest pusta i zostanie utworzona dynamicznie przez skrypt. Jest to możliwe dzięki temu, że zdarzeniu onload zostało przypisane wywołanie funkcji createDiv. A zatem funkcja ta zostanie wywołana po załadowaniu strony przez przeglądarkę.

Co się dzieje w funkcji createDiv? Ma utworzyć warstwę, która zawiera pewien tekst oraz reaguje na kliknięcia. Za pomocą metody createElement jest więc tworzony nowy element strony:

```
var divEl = document.createElement("div");
```

Następnie tworzona jest jego właściwość `onclick` i jej przypisywana jest funkcja anonimowa, będąca procedurą obsługi zdarzenia `onclick`:

```
divEl.onclick = function(){}
```

Działa tak samo jak w przykładzie z listingu 4.26.

W dalszej części kodu tworzony jest węzeł tekstowy, który stanie się treścią warstwy `divEl`:

```
var divElTextNode = document.createTextNode(str);
```

Jest dodawany do tej warstwy za pomocą metody `appendChild`:

```
divEl.appendChild(divElTextNode);
```

To postępowanie znamy z lekcji 13.

Na zakończenie warstwa wraz z węzłem tekstowym jest dodawana do treści strony (sekcji `body`). Odpowiada za to instrukcja:

```
document.body.appendChild(divEl);
```

(jak pamiętamy — tabela 4.4 — obiekt `document` posiada właściwość `body`, która jest obiektem zawierającym treść strony — treść sekcji `<body>`).

Po wczytaniu strony do przeglądarki zobaczymy, że zawiera ona warstwę z przykładowym tekstem oraz reaguje na kliknięcia, mimo iż w sekcji `body` nie zdefiniowaliśmy żadnej treści. A więc faktycznie dynamicznie tworzonym elementom witryny można przypisywać procedury obsługi zdarzeń.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 14.1.

Zmień kod z listingu 4.20 tak, aby pierwsze kliknięcie zmieniało tekst znajdujący się na warstwie, drugie przywracało jego oryginalną zawartość, trzecie ponownie go zmieniało itd.

Ćwiczenie 14.2.

Popraw kod z listingu 4.26 i 4.27 tak, by nie występowało niepotrzebne wielokrotne przypisywanie procedury obsługi zdarzenia `onclick` warstwy `div2`. Skrypt powinien wykryć, czy istnieje taka procedura, a jeśli tak, nie dokonywać ponownego przypisania.

Ćwiczenie 14.3.

Napisz kod witryny zawierającej po wczytaniu do przeglądarki jedną warstwę. Kliknięcie tej warstwy powinno spowodować utworzenie i wyświetlenie drugiej, również reagującej na kliknięcia.

Ćwiczenie 14.4.

Umieść na stronie warstwę zawierającą przykładowy tekst (przysłowie, tekst reklamowy, poradę dnia itp.). Każde kliknięcie warstwy powinno powodować zmianę widocznego na niej tekstu na inny. Wszystkie teksty (minimum 3) powinny być zapisane w skrypcie (generowane przez skrypt).

Lekcja 15. Elementy witryny

W lekcji 15. zajmiemy się elementami witryny, takimi jak przyciski, pola wyboru czy listy rozwijane. Pierwotnie były one elementami formularzy HTML, jednak od dawna z powodzeniem mogą funkcjonować na stronie jako samodzielne elementy. Zobaczmy zatem, jak je umieszczać na stronie, jakie mają właściwości i metody, i jak je obsługiwać z poziomu JavaScriptu. Do obsługi elementów użyjemy technik poznanych w lekcji 14. Każdy przykład będzie zawierał, oprócz kodu (X)HTML, procedurę obsługi konkretnego zdarzenia (najczęściej `onclick`) w postaci funkcji JavaScript.

Elementy typu <input>

Większość opisywanych w tej lekcji elementów witryny jest generowana przez znacznik `<input>` z różnymi wartościami atrybutu `type` (np. parametr `type` ustawiony na `button` oznacza przycisk). W związku z tym, elementy te, gdy odwołujemy się do nich w kodzie JavaScript, mają wiele wspólnych właściwości i metod. Aby więc niepotrzebnie nie powtarzać ich opisów, zostały zebrane w tabelach 4.12 (właściwości) i 4.13 (metody). Uwzględnione zostały tylko te składowe obiektów generowanych za pomocą znacznika `<input>`, które są obsługiwane przez wszystkie przeglądarki (Firefox, Internet Explorer, Opera) oraz zgodne ze standardami W3C. W kolumnie *Dotyczy* znajduje się lista wartości atrybutów `type`, których dotyczy dana właściwość bądź metoda. Dodatkowo uwzględnione zostały również elementy generowane przez znaczniki `<textarea>` i `<option>`.

Tabela 4.12. Właściwości obiektów typu <input>

Właściwość	Opis	Dotyczy
<code>accessKey</code>	Pobiera lub ustala klawisz pozwalający na dostęp do elementu.	<code>button, checkbox, radio, text, textarea,</code>
<code>alt</code>	Pobiera lub ustala tekst alternatywny, wyświetlany, gdy przeglądarka nie obsługuje danego elementu.	<code>button, checkbox, radio, text,</code>
<code>checked</code>	Określa, czy dany element ma być zaznaczony.	<code>checkbox, radio,</code>
<code>className</code>	Pobiera lub ustala wartość atrybutu <code>class</code> .	<code>button, checkbox, radio, text, textarea, select, option</code>

Tabela 4.12. Właściwości obiektów typu <input> (ciąg dalszy)

Właściwość	Opis	Dotyczy
cols	Pobiera lub ustala szerokość pola tekstowego.	textarea
defaultChecked	Zwraca wartość domyślną atrybutu checked.	checkbox, radio,
defaultValue	Pobiera lub ustala wartość domyślną elementu.	text, textarea
dir	Pobiera lub ustala określenie kierunku tekstu.	button, checkbox, radio, text, textarea, select, option
disabled	Określa, czy dany element ma być włączony.	button, checkbox, radio, text, textarea, select, option
form	Zawiera odniesienie do formularza, w którym znajduje się dany element.	button, checkbox, radio, text, textarea, select, option
id	Pobiera lub ustala identyfikator (atribut id) elementu.	button, checkbox, radio, text, textarea, select, option
maxLength	Pobiera lub ustala maksymalną liczbę znaków w polu tekstowym.	text
lang	Pobiera lub ustala kod języka danego elementu.	button, checkbox, radio, textarea, select, option
length	Zawiera liczbę opcji zawartych na liście.	select
multiple	Określa, czy lista jest listą wielokrotnego wyboru.	select
name	Pobiera lub ustala nazwę elementu.	button, checkbox, radio, textarea, select
selectedIndex	Pobiera lub ustala indeks aktywnej opcji.	select
readOnly	Określa, czy pole tekstowe ma być polem tylko do odczytu.	text, textarea
rows	Pobiera lub ustala wysokość pola tekstowego.	textarea
size	Pobiera lub ustala rozmiar pola tekstowego bądź listę elementów widocznych na liście.	text, select
tabIndex	Pobiera lub ustala indeks elementu używany przy przemieszczaniu się po elementach strony za pomocą klawisza Tab.	button, checkbox, radio, textarea, select
title	Pobiera lub ustala treść podpowiedzi odnoszącej się do danego elementu.	button, checkbox, radio, textarea, select, option
type	Zawiera określenie typu danego elementu.	button, checkbox, radio, textarea, select
value	Pobiera lub ustala wartość elementu (atribut value).	button, checkbox, radio, textarea, option

Tabela 4.13. Metody obiektów typu <input>

Metoda	Opis	Dotyczy
blur()	Usuwa fokus z elementu.	button, checkbox, radio, textarea
click()	Symuluje kliknięcie elementu (nie wywołuje procedury obsługi zdarzenia onclick).	button, checkbox, radio, textarea
focus()	Ustawia fokus na elemencie.	button, checkbox, radio, textarea
remove()	Usuwa element z listy.	select

Przyciski

Przycisk można zdefiniować za pomocą znacznika <input> z parametrem type ustawionym na wartość button. Natomiast znajdujący się na przycisku tekst może być zdefiniowany za pomocą atrybutu value. Przyciski obsługują większość zdarzeń, w tym najważniejsze dla tego elementu, czyli onclick, które pozwala wykryć kliknięcie. Ogólna definicja znacznika <input> tworzącego ten element witryny jest następująca:

```
<input
    type = "button"
    name = "nazwa"
    value = "wartość"
    [disabled]
    [onzdarzenie="obsługa zdarzenia"]
/>
```

I tak, *nazwa* to identyfikator, dzięki któremu możemy się później w prosty sposób do tego elementu odwoływać (ma znaczenie w przypadku umieszczenia przycisku w formularzu). Parametr *wartość* to tekst, który będzie widoczny na przycisku. Jeśli do przycisku będziemy się odwoływać w kodzie skryptu, w definicji należy również uwzględnić atrybut id. Jak w prosty sposób użyć przycisku, obrazuje kod widoczny na listingu 4.29.

Listing 4.29. Przycisk reagujący na kliknięcia

```
<body>
<div id="div1">
<input type="button"
       name="przycisk1"
       value="Kliknij mnie!"
       onclick="alert('Przycisk został kliknięty!')"
>
</div>
</body>
```

Przycisk został umieszczony na stronie za pomocą znacznika <input> z atrybutem type ustawionym na button. Została mu też przypisana procedura obsługi zdarzenia onclick, którą jest bezpośrednie wywołanie metody alert. Dlatego też każde kliknięcie

przycisku spowoduje, że na ekranie pojawi się okno dialogowe ze zdefiniowanym tekstem. Wygląd standardowego przycisku zależy od przeglądarki, którą zastosowaliśmy. Przykładowo w przeglądarce Opera 9 będzie miał postać zaprezentowaną na rysunku 4.23.

Rysunek 4.23.
Przycisk
w przeglądarce Opera



Wykonajmy jeszcze jeden przykład. Umieścimy na stronie przycisk zawierający pewien napis. Kliknięcie przycisku spowoduje zmianę tego napisu, kolejne kliknięcie — przywrócenie pierwotnego tekstu itd. Kod HTML sekcji body został zaprezentowany na listingu 4.30.

Listing 4.30. Przycisk z przypisaną procedurą obsługi zdarzenia onclick

```
<body>
  <div id="div1">
    <input type="button"
      name="przycisk1"
      id="btn1"
      value="Kliknij mnie!"
      onclick="btn1Click();"
    >
  </div>
</body>
```

Definicja przycisku jest standardowa i podobna do tej z listingu 4.29. Użyty został dodatkowy atrybut — id — który pozwoli na odwołanie się do przycisku w kodzie skryptu. Inaczej wygląda też atrybut onclick. Tym razem procedurą obsługi zdarzenia onclick jest funkcja o nazwie btn1Click. Treść tej funkcji należy umieścić w skrypcie znajdującym się w sekcji head. Przyjmie on postać widoczną na listingu 4.31.

Listing 4.31. Skrypt obsługujący kliknięcia przycisku

```
<script type="text/javascript">
  var btn1Tekst = "Jeszcze raz proszę!";
  function btn1Click()
  {
    var btn1 = document.getElementById("btn1");
    var tempTekst = btn1.value;
    btn1.value = btn1Tekst;
    btn1Tekst = tempTekst;
  }
</script>
```

W skrypcie została zadeklarowana zmienna globalna btn1Tekst. Jej zadaniem jest przechowywanie tekstu, który ma się znaleźć na przycisku po jego kliknięciu. Początkowo jest to tekst Jeszcze raz proszę!. Procedurą obsługi zdarzenia onclick przycisku jest funkcja btn1Click. Musi ona wykonać dwa zadania. Po pierwsze, powinna zapamiętać aktualnie znajdujący się na przycisku tekst (tak aby mógł być przypisany przy kolejnym kliknięciu), po drugie — zmienić znajdujący się na przycisku tekst na poprzednio zapamiętany.

Najpierw za pomocą metody getElementById pobiera odwołanie do przycisku btn1, a następnie przypisuje aktualnie znajdujący się na nim tekst zmiennej pomocniczej tempTekst:

```
var tempTekst = btn1.value;
```

Później właściwości value obiektu btn1 (czyli przycisku) przypisuje wartość zmiennej globalnej btn1Tekst:

```
btn1.value = btn1Tekst;
```

Tym samym zmienia tekst na poprzednio zapamiętany. W ostatnim kroku przypisuje zmiennej btn1Tekst wartość zmiennej tempTekst:

```
btn1Tekst = tempTekst;
```

A zatem zapamiętuje w zmiennej btn1Tekst napis, który ma się pojawić przy kolejnym kliknięciu.

Tym samym, po uruchomieniu strony użytkownik zobaczy przycisk, na którym znajduje się napis Kliknij mnie!. Gdy kliknie ten przycisk, tekst zmieni się na: Jeszcze raz proszę!. Ponowne kliknięcie spowoduje przywrócenie tekstu oryginalnego itd.

Pola wyboru typu checkbox

Pola wyboru typu checkbox umieszczamy na witrynie za pomocą znacznika <input> z parametrem type ustawionym na checkbox. Schematycznie taka konstrukcja ma postać:

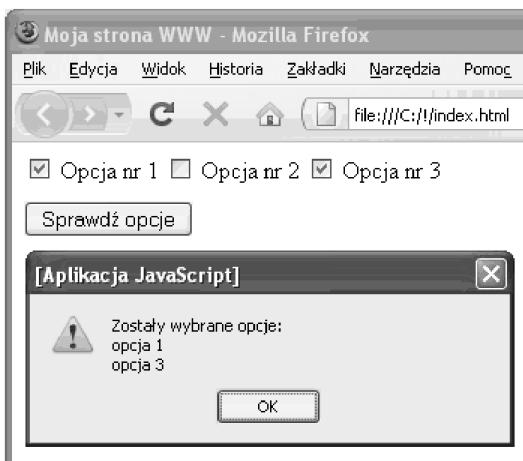
```
<input  
    type="checkbox"  
    name="nazwa"  
    value="wartość"  
    [checked]  
    [disabled]  
    [onzdarzenie="obsługa zdarzenia"]  
/>
```

Oprócz typowych atrybutów (name, value), których znaczenie jest takie same jak w przypadku przycisków, do dyspozycji mamy dodatkowo atrybut checked, określający, czy dane pole ma być początkowo zaznaczone (atybut checked obecny), czy też nie (brak atrybutu checked).

Pół typu checkbox używamy, gdy chcemy dać użytkownikowi witryny możliwość wybrania pewnych opcji. Sprawdźmy więc, jak odczytywać stan tych elementów w skrypcie JavaScript. Naszym zadaniem będzie utworzenie witryny zawierającej kilka pól typu checkbox reprezentujących różne opcje oraz przycisk. Kliknięcie przycisku ma spowodować wyświetlenie informacji o tym, które z opcji zostały wybrane przez użytkownika — jest to widoczne na rysunku 4.24. Najpierw przygotujemy kod (X)HTML witryny. Treść sekcji body została przedstawiona na listingu 4.32.

Rysunek 4.24.

Wyświetlona
została informacja
o zaznaczonych
opcjach



Listing 4.32. Kod witryny zawierającej pola wyboru typu checkbox

```
<body>
    <div id="div1" style="margin-bottom:10px;">
        <input type="checkbox"
            value="1"
            id="chb1" /> Opcja nr 1
        <input type="checkbox"
            value="2"
            id="chb2" /> Opcja nr 2
        <input type="checkbox"
            value="3"
            id="chb3" /> Opcja nr 3
    </div>
    <div id="div2">
        <input type="button" name="przycisk1"
            id="btn1" onclick="btn1Click();"
            value="Sprawdź opcje"
        >
    </div>
</body>
```

Witryna zawiera dwie warstwy. Na pierwszej znalazły się pola wyboru, a na drugiej — przycisk. Pola zostały zdefiniowane za pomocą znaczników `<input>` z atrybutem `type` ustawionym `checkbox`. Każde z nich otrzymało swój identyfikator (odpowiednio `chb1`, `chb2` i `chb3`), a także wartość (odpowiednio 1, 2 i 3). Wartość danego pola jest

definiowana za pomocą atrybutu value, a kodzie skryptu można ją odczytać, odwołując się do właściwości value (tabela 4.12). Obok każdego pola zostało umieszczone tekst opisujący opcję, którą ono reprezentuje.

Znajdujący się na drugiej warstwie przycisk ma spowodować wywołanie procedury odczytującej stan pól i wyświetlającej informację, które z nich są zaznaczone. Dlatego też w zdarzeniu onclick została przypisana procedura obsługi w postaci funkcji btn1Click. Treść tej funkcji umieścimy w kodzie skryptu zdefiniowanym w sekcji <head> za pomocą znacznika <script> (listing 4.33).

Listing 4.33. Funkcja określająca stan pól wyboru

```
<script type="text/javascript">
    function btn1Click()
    {
        var komunikat = "";
        var chb1 = document.getElementById('chb1');
        var chb2 = document.getElementById('chb2');
        var chb3 = document.getElementById('chb3');
        if(!chb1.checked & !chb2.checked &
           !chb3.checked){
            komunikat = "Nie została wybrana żadna opcja.";
        }
        else{
            komunikat = "Zostały wybrane opcje:";
            if(chb1.checked)
                komunikat += "\nopcja " + chb1.value;
            if(chb2.checked)
                komunikat += "\nopcja " + chb2.value;
            if(chb3.checked)
                komunikat += "\nopcja " + chb3.value;
        }
        alert(komunikat);
    }
</script>
```

Na początku funkcji btn1Click została utworzona zmienna komunikat. Będzie ona przechowywała komunikat, który w efekcie działania skryptu ma się pojawić na ekranie. Następnie za pomocą doskonale znanych już konstrukcji o postaci:

```
var chbX = document.getElementById('chbX');
```

zostały pobrane odwołania do wszystkich pól wyboru. W ten sposób powstały obiekty chb1, chb2 i chb3 odpowiadające polom o takich samych identyfikatorach.

Pierwsza instrukcja warunkowa if bada stan właściwości checked każdego pola. Właściwość ma wartość true, jeśli dane pole jest zaznaczone, lub false, jeśli nie jest. To oznacza, że złożony warunek:

```
!chb1.checked && !chb2.checked && !chb3.checked
```

jest prawdziwy, gdy żadne z pól nie jest zaznaczone. W takiej sytuacji zmiennej komunikat przypisujemy taką informację.

Jeżeli jednak warunek jest fałszywy, czyli gdy przynajmniej jedno z pól jest zaznaczone, wykonywana jest część kodu w bloku else. Najpierw zmiennej komunikat przypisywana jest wspólna część komunikatu, niezależna od tego, które z pól są zaznaczone. Następnie za pomocą serii instrukcji warunkowych badany jest stan właściwości checked każdego z pól. W każdym przypadku, gdy jest ona równa true, do zmiennej komunikat dopisywany jest numer opcji pobrany z właściwości value.

Na zakończenie zmienna komunikat jest używana jako argument w wywołaniu metody alert, co powoduje wyświetlenie przygotowanego komunikatu w osobnym oknie dialogowym.

Pola wyboru typu radio

Pola wyboru typu radio definiuje się w podobny sposób jak pola typu checkbox. Podobne jest też ich zadanie — pozwalają na wybór opcji. Co prawda, można ich używać dokładnie tak samo jak pól checkbox, ale najczęściej stosuje się je w przypadku opcji wykluczających, czyli takich, gdzie w danym momencie może być aktywna tylko jedna. Wystarczy utworzyć grupę pól typu radio, aby zaznaczenie jednego automatycznie powodowało usunięcie zaznaczenia poprzedniego. Grupę tworzymy, przypisując zestawowi pól taką samą wartość atrybutu name. Ogólna definicja jest następująca:

```
<input  
    type="radio"  
    name="nazwa"  
    value="wartość"  
    [checked]  
    [disabled]  
    [onchange="obsługa zdarzenia"]  
>
```

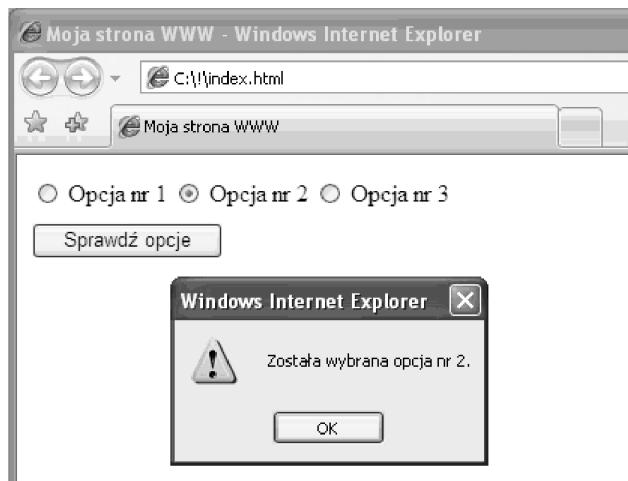
Nie różni się więc znacząco od typu checkbox.

Wykonajmy przykład podobny do poprzedniego, ale korzystający z pól typu radio. Na stronie umieścimy wzajemnie się wykluczające trzy pola tego typu. Będą one tworzyć jedną grupę i w jednej chwili będzie mogło być zaznaczone tylko jedno nich. Pod polami znajdzie się przycisk, którego kliknięcie spowoduje wyświetlenie informacji, które pole jest zaznaczone, co jest widoczne na rysunku 4.25. Kod (X)HTML sekcji body został zaprezentowany na listingu 4.34.

Ogólna struktura kodu jest taka sama jak kodu z listingu 4.32. W sekcji body znajdują się dwie warstwy. Na pierwszej umieszczone zostały pola wyboru, a na drugiej — przycisk. Przyciskowi została przypisana procedura obsługi zdarzenia onclick w postaci funkcji btn1Click. Pola wyboru zostały zdefiniowane za pomocą znacznika <input> z parametrem type ustawionym na radio. Każde pole ma własny identyfikator (wartość atrybutu id — odpowiednio rb1, rb2 i rb3) oraz wartość (wartość atrybutu value — odpowiednio 1, 2 i 3). Wszystkie pola mają natomiast tę samą wartość atrybutu name — grupa1 — dzięki czemu tworzą jedną grupę i są polami wzajemnie się wykluczającymi.

Rysunek 4.25.

Wyświetlenie informacji o tym, które pole zostało zaznaczone

**Listing 4.34.** Strona zawierająca pola wyboru typu radio

```
<body>
<div id="div1" style="margin-bottom:10px;">
<input type="radio"
      value="1"
      name="grupa1"
      id="rb1" /> Opcja nr 1
<input type="radio"
      value="2"
      name="grupa1"
      id="rb2" /> Opcja nr 2
<input type="radio"
      value="3"
      name="grupa1"
      id="rb3" /> Opcja nr 3
</div>
<div>
<input type="button" name="przycisk1"
       id="btn1" onclick="btn1Click();"
       value="Sprawdź opcje"
    >
</div>
</body>
```

Stwierdzeniem, które pole zostało zaznaczone przez użytkownika witryny, zajmuje się funkcja `btn1Click`. Jej treść została zaprezentowana na listingu 4.35.

Listing 4.35. Kod funkcji obsługującej pola typu radio

```
<script type="text/javascript">
function btn1Click(){
  var komunikat = "Została wybrana opcja nr ";
  var rb1 = document.getElementById('rb1');
  var rb2 = document.getElementById('rb2');
  var rb3 = document.getElementById('rb3');
```

```
if(rb1.checked){  
    komunikat += rb1.value;  
}  
else if(rb2.checked){  
    komunikat += rb2.value;  
}  
else if(rb3.checked){  
    komunikat += rb3.value;  
}  
else{  
    komunikat = "Nie została wybrana żadna opcja";  
}  
komunikat += ".  
alert(komunikat);  
}  
</script>
```

Początek funkcji przypomina funkcję z listingu 4.33. Tworzona jest zmienna komunikat oraz pobierane są odwołania do poszczególnych pól. Odwołania są zapisywane w zmiennych rb1, rb2 i rb3. Zmienna komunikat otrzymuje natomiast od razu ciąg znaków Została wybrana opcja nr, który zostanie uzupełniony po zbadaniu stanu pól.

Stan pól jest badany za pomocą złożonej instrukcji warunkowej if...else if. W każdym kroku sprawdzany jest stan właściwości checked kolejnego pola (zmienne rb1, rb2 i rb3). Czynność ta może być wykonywana w ten sposób, bowiem w przypadku pól tworzących jedną grupę tylko jedno z nich (lub żadne) może być zaznaczone. Jeżeli żadna z właściwości checked nie ma wartości true, oznacza to, że żadne pole nie zostało zaznaczone. W takiej sytuacji zmienia się komunikat przypisany zmiennej komunikat.

Na końcu kodu do zmiennej komunikat jest dodawany znak kropki (by zachować reguły interpunkcji). Zmienna jest też używana jako argument w wywołaniu metody alert.

Zwykłe pola tekstowe

Pola tekstowe umożliwiają użytkownikowi strony WWW wprowadzanie danych. Wartość wpisana do pola może być bez problemów odczytana przez skrypt. Pole jest definiowane za pomocą znacznika <input> z parametrem type ustawionym na text. Schematycznie wygląda to tak:

```
<input  
    type="text"  
    name="nazwa"  
    value="wartość"  
    size="rozmiar"  
    maxlength="maksymalna długość"  
    [readonly]  
    [disabled]  
    [onzdarzenie="obsługa zdarzenia"]  
/>
```

Oprócz typowych atrybutów znanych z wcześniejszych opisów, znajduje się tu też kilka nowych. I tak `size` określa w znakach rozmiar pola, a `maxlength` — maksymalną liczbę znaków, które będzie można wprowadzić do pola. Użycie atrybutu `readonly` spowoduje, że pole będzie mogło być tylko odczytywane.

Sprawdźmy, jak pole tekstowe zachowuje się w praktyce. Umieścimy jedno na witrynie. Umożliwi ono wprowadzenie wyrażenia arytmetycznego. Kliknięcie znajdującego się obok przycisku spowoduje obliczenie wyrażenia i wyświetlenie jego wyniku, co zostało pokazane na rysunku 4.26. Kod (X)HTML sekcji body takiej witryny został zaprezentowany na listingu 4.36.

Rysunek 4.26.

Pole tekstowe użyte do obliczania wartości wyrażeń



Listing 4.36. Witryna z polem tekstowym

```
<body>
<div id="div1" style="margin-bottom:10px;">
    <input type="text"
        value="Wprowadź wyrażenie"
        id="tf1"
    />
    <input type="button" name="przycisk1"
        id="btn1" onclick="btn1Click();"
        value="Oblicz"
    />
</div>
<div id="div2">
    Tutaj pojawi się wynik obliczeń
</div>
</body>
```

Strona zawiera pole tekstowe zdefiniowane za pomocą znacznika `<input>` z parametrem `type` ustawionym na `text`. Polu został nadany identyfikator `tf1`, a także wartość atrybutu `value`. Wartość ta zostanie wyświetlona w polu po załadowaniu strony do przeglądarki. Obok znajdująemy przycisk, którego kliknięcie spowoduje obliczenie wyrażenia wprowadzonego do pola i wyświetlenie wyników na warstwie `div2`. Treść funkcji wykonującej obliczenia została przedstawiona na listingu 4.37.

Listing 4.37. Funkcja obliczająca wyrażenie z pola tekstowego

```
<script type="text/javascript">
    function btn1Click(){
        var tf1 = document.getElementById('tf1');
        var div2 = document.getElementById('div2');
        var wyrażenie = tf1.value;
        try{
            var wynik = eval(wyrażenie);
        }
        catch(e){
            var wynik = "Wyrażenie jest niepoprawne.";
        }
        div2.innerHTML = wynik;
    }
</script>
```

Funkcja najpierw pobiera odwołania do pola tekstowego tf1 oraz warstwy div2 i zapisuje je w zmiennych o takich samych nazwach. Następnie odczytuje wartość właściwości value pola, czyli wartość wprowadzoną przez użytkownika do pola:

```
var wyrażenie = tf1.value;
```

W ten sposób w zmiennej wyrażenie znajdzie się ciąg znaków reprezentujący pewne wyrażenie. Trzeba je więc obliczyć. Do tego celu najprościej użyć poznanej w lekcji 7. funkcji eval. Należy jednak wziąć pod uwagę, że wpisany do pola tekst wcale nie musi być prawidłowym wyrażeniem arytmetycznym, a więc funkcja eval może nie zadziałać prawidłowo. Dlatego też jej wywołanie zostało ujęte w blok try...catch. Dzięki temu skrypt zadziała właściwie nawet wtedy, jeśli wyrażenie nie będzie prawidłowe.,

Jeżeli więc ciąg znaków odczytany z pola tekstowego tf1, a znajdujący się w zmiennej wyrażenie, jest wyrażeniem prawidłowym, które może być przetworzone przez JavaScript, zostanie wykonana instrukcja:

```
var wynik = eval(wyrażenie);
```

W przeciwnym przypadku — instrukcja:

```
var wynik = "Wyrażenie jest niepoprawne.";
```

Na zakończenie wartość zmiennej wynik jest przypisywana właściwości innerHTML warstwy div2, co sprawia, że pojawia się na ekranie.

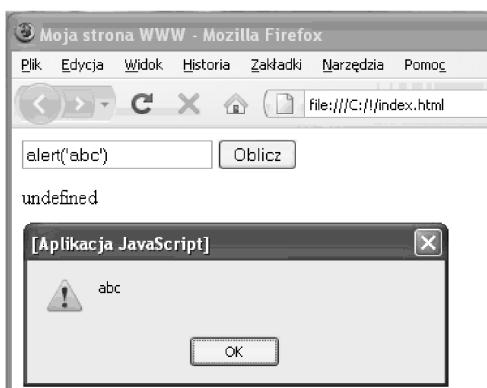
Warto przypomnieć, że funkcja eval przetwarza nie tylko wyrażenia arytmetyczne. Może również spowodować wykonanie instrukcji JavaScript, np. wywołanie funkcji. Tak też będzie w tym przypadku. Jeśli w polu tekstowym wpiszemy np. instrukcję:

```
alert('abc')
```

to po kliknięciu przycisku *Oblicz* na ekranie pojawi się okno dialogowe, tak jak zostało to zaprezentowane na rysunku 4.27. A zatem nasz skrypt dodatkowo pozwala na wykonywanie instrukcji JavaScriptu.

Rysunek 4.27.

Instrukcja
wprowadzona do pola
została wykonana



Rozszerzone pola tekstowe

O ile zwykłe pole tekstowe pozwalało na wprowadzenie tekstu w jednym wierszu, o tyle pole rozszerzone umożliwia wprowadzenie wielu wierszy. Jest ono definiowane za pomocą znacznika <textarea> o schematycznej postaci:

```
<textarea
  name="nazwa pola"
  rows="wiersze"
  cols="kolumny"
  [readonly]
  [disabled]
  [onzdarzenie="obsługa zdarzenia"]
>pierwotny tekst</textarea>
```

gdzie `rows` określa wysokość (liczbę wierszy) pola, a `cols` — jego szerokość (liczbę kolumn). Użycie atrybutu `readonly` spowoduje, że pole będzie mogło być tylko odczytywane.

W kolejnym przykładzie zobaczymy, jak użyć rozszerzonego pola tekowego. Umieszcimy je na stronie, a wraz z nim przycisk i dodatkową warstwę. Kliknięcie przycisku będzie powodowało przepisanie tekstu wprowadzonego do pola na warstwę. Kod (X)HTML został zaprezentowany na listingu 4.38.

Listing 4.38. Witryna zawierająca rozszerzone pole tekstowe

```
<body>
  <div>
    <textarea cols="30" rows="4" id="ta1"></textarea>
  </div>
  <div>
    <input type="button" name="przycisk1"
      id="btn1" onclick="btn1Click();"
      value="Przepisz na stronę"
    />
  </div>
  <div id="div3"></div>
</body>
```

Kod sekcji body składa się w sumie z trzech warstw. Pierwsza zawiera pole tekstowe zdefiniowane za pomocą znacznika <textarea>. Jego szerokość została określona na 30 kolumn, a wysokość na 4 wiersze. Został mu również nadany identyfikator ta1, który pozwoli na odwoływanie się do pola w kodzie skryptu.

Druga warstwa zawiera przycisk zdefiniowany za pomocą znacznika <input> z parametrem type ustawionym na button. Przyciskowi została przypisana procedura obsługi zdarzenia onclick w postaci funkcji o nazwie btn1Click. Na przycisku będzie widniał ciąg znaków Przepisz na stronę.

Trzeciej warstwie został nadany identyfikator div3. Początkowo jest pusta, będzie się na niej pojawiać tekst wprowadzany przez użytkownika do pola tekstopowego ta1. Za przeniesienie tekstu z pola na warstwę odpowiada funkcja btn1Click. Jej treść została przedstawiona na listingu 4.39.

Listing 4.39. Funkcja przenosząca tekst z pola tekstopowego na warstwę

```
<script type="text/javascript">
    function btn1Click()
    {
        var ta1 = document.getElementById('ta1');
        var div3 = document.getElementById('div3');
        var tekst = ta1.value;
        if(tekst != "")
            div3.innerHTML += tekst;
    }
</script>
```

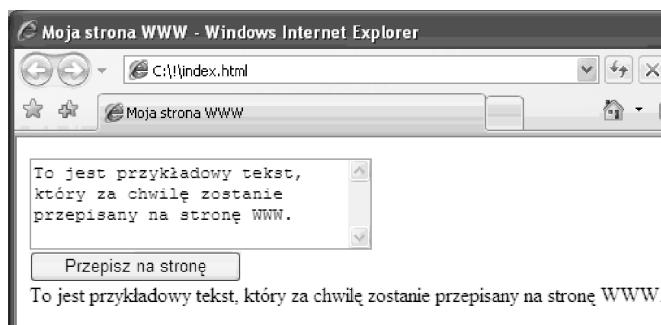
Funkcja za pomocą metody getElementById pobiera odwołania do pola tekstopowego ta1 i warstwy div3, a następnie odczytuje zawarty w polu tekst. Wykonuje to, odwołując się do właściwości value pola, które jest reprezentowane przez obiekt ta1. Odczytana wartość zapisuje w zmiennej tekst i za pomocą instrukcji warunkowej if bada, czy nie jest ona pustym ciągiem znaków. Jeśli nie jest, dopisuje do właściwości innerHTML warstwy div3 wartość zmiennej tekst (czyli tekst znajdujący się w polu). A zatem jeśli w polu tekstopowym ta1 znajduje się jakiś napis, po pierwszym kliknięciu przycisku pojawi się w treści witryny na warstwie div3, co zostało pokazane na rysunku 4.28. Jeżeli natomiast w polu nie będzie żadnych danych, kliknięcie przycisku nie spowoduje żadnej reakcji. Warto samodzielnie zastanowić się, czy w skrypcie jest potrzebna użyta w nim instrukcja warunkowa, czy też można ją usunąć bądź umieścić w innym miejscu.

Pola tekstowe typu password

Pole tekstopowe typu password jest bardzo podobne do pola typu text, a obsługa ich jest wręcz taka sama. Różnica polega na tym, że w polu typu password tekst w trakcie wprowadzania nie jest wyświetlany, a zamiast niego pojawiają się znaki maskujące (kropki, gwiazdki itp. — rysunek 4.30). Pole to nadaje się zatem do wprowadzania wszelkiego rodzaju haseł i kodów dostępu. Definiujemy je za pomocą znacznika <input> z parametrem type ustawionym na password, schematycznie:

Rysunek 4.28.

Tekst z rozszerzonego pola tekowego został przepisany na warstwę



```
<input
    type="password"
    name="nazwa"
    value="wartość"
    size="rozmiar"
    maxlength="maksymalna długość"
    [readonly]
    [disabled]
    [onzdarzenie="obsługa zdarzenia"]>
/>
```

Kiedy użyć pola typu password? Przyda się, gdy będziemy wymagać od użytkownika witryny podania kodu dostępu. Tego typu zadania z reguły wymagają uwierzytelniania po stronie serwera, ale sposób wykorzystania tego pola można pokazać na prostszym przykładzie. Założmy, że część witryny ma być dostępna tylko po podaniu hasła. Umieścimy więc na niej pole typu password oraz przycisk, co zostało zaprezentowane na rysunku 4.29. Generujący taką stronę kod (X)HTML znajduje się na listingu 4.40.

Rysunek 4.29.

Pole typu password

**Listing 4.40.** Witryna zawierająca pole typu password

```
<body>
<div style="margin-bottom:10px;">
    Wprowadź kod:
    <input type="password" id="pf1">
    <input type="button" name="przycisk1"
        id="btn1" onclick="btn1Click();"
        value="Wejdź"
    />
</div>
<div id="div2"></div>
</body>
```

Strona zawiera dwie warstwy. Na pierwszej umieszczone jest pole tekstowe typu password, któremu został nadany identyfikator pf1 oraz przycisk o identyfikatorze btn1. Przyciskowi została przypisana procedura obsługi zdarzenia onclick w postaci funkcji btn1Click. Druga warstwa początkowo jest pusta. Na niej będzie wyświetlana część witryny zabezpieczona hasłem.

Treść funkcji btn1Click, badającej poprawność wprowadzonego przez użytkownika kodu dostępu, znajduje się na listingu 4.41.

Listing 4.41. Funkcja badająca poprawność hasła

```
<script type="text/javascript">
    function btn1Click()
    {
        var pf1 = document.getElementById('pf1');
        var div2 = document.getElementById('div2');
        var tekst = "Te dane są dostępne po podaniu kodu.";
        if(pf1.value == "tajny_kod"){
            div2.innerHTML = tekst;
        }
        else{
            alert("Podany kod jest nieprawidłowy!");
        }
    }
</script>
```

Najpierw są pobierane oraz zapisywane w zmiennych pf1 i div2 odwołania do pola typu password o identyfikatorze pf1, a także do warstwy o identyfikatorze div2. Natomiast zmienna tekst przechowuje treść, która ma się pojawić na warstwie div2 po podaniu poprawnego hasła. Za pomocą instrukcji warunkowej if badane jest, czy treść wprowadzona do pola (wartość właściwości value) jest zgodna z prawidłowym kodem (w tym przypadku z ciągiem znaków tajny_kod). Jeśli jest, właściwości innerHTML przypisywana jest wartość zmiennej tekst (tym samym pojawia się na stronie). Jeśli nie, wyświetlane jest okno dialogowe informujące o tym, że został podany błędny kod.

Oczywiście, taki sposób autoryzacji nie stanowi dobrego zabezpieczenia witryny, gdyż zarówno hasło, jak i chroniona treść są widoczne w kodzie skryptu i każdy może je podejrzeć. W praktyce autoryzacja powinna odbywać się na serwerze, przy użyciu innych technik, np. PHP. Jest to jednak temat wykraczający poza ramy tej książki⁸.

Listy wyboru

Lista umożliwiająca wybór zdefiniowanych opcji jest tworzona za pomocą znaczników `<select>` i `<option>`. Może to być lista rozwijana bądź o stałej wielkości. W drugim przypadku może to być lista jednokrotnego wyboru bądź wyboru wielokrotnego. Ogólny schemat budowy listy jest następujący:

⁸ Osobom zainteresowanym językiem PHP i tworzeniem skryptów działających po stronie serwera WWW można polecić książkę *PHP5. Praktyczny kurs* (<http://helion.pl/ksiazki/php5pk.htm>), a także *PHP 101 praktycznych skryptów. Wydanie II* (<http://helion.pl/ksiazki/php102.htm>).

```

<select
  name="nazwa"
  [size="rozmiar"]
  [multiple]
  [disabled]
  [onzdarzenie="obsługa zdarzenia"]
>
  <option value="wartość1" [selected]> opcja1</option>
  <option value="wartość2" [selected]> opcja2</option>
  ...
  <option value="wartośćN" [selected]> opcjaN</option>
</select>

```

gdzie *rozmiar* jest liczbą pozycji na liście, które mają być wyświetlane, a za pomocą znaczników `<option>` określa się wartości na liście. Dodatkowy parametr `selected` oznacza, że dana pozycja ma być domyślnie zaznaczona. Jeżeli parametr *rozmiar* ma wartość 1, powstaje lista rozwijana, a gdy jest większy od 1 — lista o stałym rozmiarze. W tym drugim przypadku podanie parametru `multiple` powoduje utworzenie listy wielokrotnego wyboru.

W listach najczęściej korzysta się ze zdarzenia `onchange`, które jest generowane, gdy zmieni się aktywna pozycja. Sprawdzimy to w praktyce. Umieścimy na stronie listę rozwijaną, zawierającą kilka przykładowych pozycji. Gdy jedna zostanie wybrana, na stronie wyświetlimy jej nazwę, indeks oraz przypisaną jej wartość. Pracę zacznijmy od kodu (X)HTML. Został zaprezentowany na listingu 4.42.

Listing 4.42. Strona zawierająca listę rozwijaną

```

<body>
  <div style="margin-bottom:10px;">
    <select
      id="listal"
      size="1"
      onchange="listalChange();"
    >
      <option value=""> Wybierz jedną z opcji</option>
      <option value="a"> So far away</option>
      <option value="b"> Money for nothing</option>
      <option value="c"> Walk of life</option>
      <option value="d"> Your latest trick</option>
      <option value="e"> Why Worry</option>
      <option value="f"> Ride across the river</option>
      <option value="g"> The man's too strong</option>
      <option value="h"> One world</option>
      <option value="i"> Brothers in arms</option>
    </select>
  </div>
  <div id="div2"></div>
</body>

```

Na stronie znajdują się dwie warstwy. Pierwsza zawiera listę zdefiniowaną za pomocą znacznika `<option>`. Liście został nadany identyfikator `listal`, a jej rozmiar (atribut `size`) został określony na 1. Oznacza to, że będzie listą rozwijaną. Zdarzeniu `onchange`

listy została przypisana procedura obsługi w postaci funkcji lista1OnChange, a to oznacza, że funkcja będzie wykonywana za każdym razem, gdy zmieni się aktywna pozycja na liście.

Poszczególne opcje zostały zdefiniowane przy użyciu znaczników <option>. Każdy z nich zawiera atrybut value określający wartość przypisaną danej opcji. W tym przypadku są to kolejne litery alfabetu, choć wartości te mogą być dowolne — takie, jakie w danym przypadku są potrzebne. Za warstwą zawierającą listę znajduje się druga, o identyfikatorze div2. Początkowo jest pusta i zostanie użyta do wyświetlenia danych.

Po wczytaniu takiego kodu do przeglądarki i rozwinięciu listy zobaczymy widok zaprezentowany na rysunku 4.30. Czas więc przygotować kod skryptu obsługującego tę witrynę. Jest widoczny na listingu 4.43.

Rysunek 4.30.

Widok listy rozwijanej umożliwiającej wybór jednej opcji



Listing 4.43. Kod funkcji obsługującej listę rozwijaną

```
<script type="text/javascript">
    function lista1Change()
    {
        var lista = document.getElementById('lista1');
        var div2 = document.getElementById('div2');

        var opcja = lista[lista.selectedIndex];

        var tekst = "Wybrana opcja: ";
        tekst += opcja.text;
        tekst += "<br /> Wartość opcji: ";
        tekst += opcja.value;
        tekst += "<br /> Indeks opcji: ";
        tekst += opcja.index;

        div2.innerHTML = tekst;
    }
</script>
```

Odwolania do listy `list1` oraz warstwy `div2`, na której pojawią się odczytane dane, są pobierane w standardowy sposób. Ważniejsza jest tu sama obsługa listy. Na początek musimy dowiedzieć się, która pozycja jest aktywna (została wybrana przez użytkownika), oraz w jaki sposób odczytać jej wartości. Indeks aktywnej pozycji odczytamy, odwołując się do właściwości `selectedIndex` obiektu listy, czyli pisząc:

```
list1.selectedIndex
```

Gdy mamy już tę informację, możemy pobrać obiekt opisujący daną pozycję. Wszystkie pozycje, jakie są zawarte na liście, znajdują się w tablicy `options`⁹. Ponieważ jednak właściwość `options` jest właściwością domyślną, aby uzyskać dostęp do danej komórki, nie musimy pisać:

```
list1.options[indeks];
```

Wystarczy:

```
list1[indeks];
```

A więc pisząc:

```
lista[list1.selectedIndex];
```

odczytamy obiekt odpowiadający wybranej przez użytkownika opcji. Obiekt ten (element zdefiniowany w kodzie (X)HTML za pomocą znacznika `<option>`), oprócz właściwości wymienionych w tabeli 4.12, ma również kilka specyficznych, charakterystycznych tylko dla niego. Zostały zebrane w tabeli 4.14.

Tabela 4.14. Właściwości obiektu `option`

Właściwość	Opis
<code>defaultSelected</code>	Zwraca wartość domyślną atrybutu <code>selected</code> .
<code>index</code>	Zwraca indeks danej opcji.
<code>selected</code>	Określa bieżącą wartość atrybutu <code>selected</code> .
<code>text</code>	Określa wartość tekstową danej opcji.

Z obiektu reprezentowanego przez zmienną `opcja`, odpowiadającego wybranej opcji, wystarczy więc odczytać wszystkie pożądane wartości, czyli:

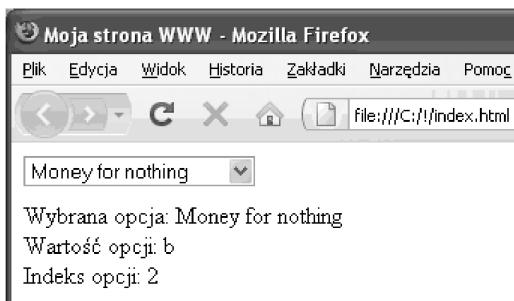
- ◆ tekst przypisany opcji (`opcja.text`),
- ◆ wartość przypisaną opcji (`opcja.value`),
- ◆ indeks opcji (`opcja.index`).

Wartości te są dopisywane do pomocniczej zmiennej `tekst`, której wartość jest z kolei przypisywana właściwości `innerHTML` warstwy `div2`. Dzięki temu cały przygotowany komunikat pojawi się na stronie, co zostało zaprezentowane na rysunku 4.31.

⁹ Dokładniej rzecz ujmując, `options` jest tzw. kolekcją (choć nietypową ze względu na konieczność zachowania kompatybilności ze starszymi wersjami przeglądarek). Jednak z powodzeniem możemy ją traktować jak tablicę.

Rysunek 4.31.

Na stronie pojawiły się dane dotyczące wybranej opcji



Jeżeli interesuje nas dodanie do listy nowej opcji, należy utworzyć obiekt typu Option, podając jako argument nazwę i, opcjonalnie, wartość danej opcji. Oto schemat:

```
new Option(text_opcji[, wartość_opcji]);
```

np.:

```
new Option("Money for nothing", "b")  
new Option("Ride across the river")
```

Taki obiekt należy następnie umieścić w tablicy options listy, np.:

```
listal.options[5] = new Option("Money for nothing", "b");  
listal[6] = new Option("Ride across the river");
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 15.1.

Zmień kod skryptu z listingu 4.33, tak by pola wyboru były polami wykluczającymi się, tzn. na raz mogło być zaznaczone tylko jedno.

Ćwiczenie 15.2.

Zmień kod przykładu z listingów 4.34 i 4.35, tak aby kliknięcie nazwy opcji powodowało zaznaczenie odpowiadającego jej pola wyboru.

Ćwiczenie 15.3.

Umieść na stronie 6 pól typu checkbox. Napisz skrypt, który uniemożliwi zaznaczenie więcej niż dwóch na raz.

Ćwiczenie 15.4.

Umieść na stronie pole tekstowe, listę i przycisk. Kliknięcie przycisku powinno spowodować dodanie do listy nowej pozycji zawierającej tekst znajdujący się w polu tekstowym.

Lekcja 16. Style CSS

Style CSS to nieodłączny element każdej nowoczesnej strony WWW. Współczesne przeglądarki pozwalają też na ich dynamiczną modyfikację. Właśnie tej tematyce w całości poświęcona jest lekcja 16. Sprawdzimy więc, jakie istnieją sposoby zmiany stylów przypisanych danym elementom strony, poznamy obiekt style i właściwość className. Zwrócimy także uwagę na występujące między przeglądarkami różnice w sposobach obsługi niektórych właściwości i interpretacji stylów.

Dostęp do atrybutów

Praktycznie każdemu elementowi strony można przypisać styl CSS — dziś trudno spotkać witryny, które ze stylów nie korzystają. Nie trzeba jednak ograniczać się tylko do statycznych przypisów występujących w kodzie (X)HTML. Jeżeli użyjemy JavaScriptu, będziemy mogli dokonywać dynamicznej zmiany stylów w trakcie działania witryny. Pierwszym sposobem modyfikacji jest skorzystanie z takich metod jak getAttribute, setAttribute i removeAttribute. Pozwalają one na pobranie, modyfikację oraz usuwanie wartości atrybutów przypisanych danemu elementowi witryny, a więc również atrybutu style zawierającego styl CSS elementu. Pobierzmy odwołanie do jakiegoś elementu strony (węzła drzewa DOM) i zapiszmy je w zmiennej obj, np.:

```
var obj = document.getElementById("pt1");
```

Gdy zechcemy dowiedzieć się, jaki mu przypisano styl, skorzystamy z metody getAttribute, np.:

```
var styl = obj.getAttribute("style");
```

Aby przypisać obiektowi styl, używamy metody setAttribute. Jej pierwszym argumentem jest nazwa atrybutu (w tym przypadku jest to style), a drugim — jego wartość. Jeżeli zatem elementowi obj chcemy przypisać styl, w którym atrybut font-weight ma wartość bold, użyjemy instrukcji:

```
obj.setAttribute("style", "font-weight:bold");
```

Usunięcie stylu osiągniemy natomiast za pomocą metody removeAttribute:

```
obj.removeAttribute("style");
```

Niestety, ten — zgodny ze standardami W3C — sposób nie działa w Internet Explorerze, ale tym problemem zajmiemy się za chwilę. Najpierw użyjmy przedstawionych metod, tak jak powinny być zastosowane. Przygotujemy stronę zawierającą warstwę, pole tekstowe oraz trzy przyciski. Przyciski będą umożliwiały:

- ◆ odczytanie aktualnych stylów przypisanych warstwie,
- ◆ przypisanie warstwie stylów wprowadzonych w polu tekstowym,
- ◆ usunięcie wszystkich stylów przypisanych warstwie.

Kod (X)HTML takiej strony został zaprezentowany na listingu 4.44.

Listing 4.44. Strona umożliwiająca manipulowanie stylami

```
<body>
  <div style="margin-bottom:10px;">
    <input type="text" id="tf1" size="30" />
  </div>
  <div style="margin-bottom:10px;">
    <input type="button" value="Odczytaj"
      onclick="odczytajStyl();"/>
    <input type="button" value="Przypisz"
      onclick="przypiszStyl();"/>
    <input type="button" value="Usuń"
      onclick="usuńStyl();"/>
  </div>
  <div id="div3"
    style="background-color:#ffff00;width:210px;height:50px;">
  </div>
</body>
```

Na stronie znajdują się trzy warstwy. Pierwsza zawiera pole tekstowe o identyfikatorze tf1, druga — trzy przyciski, trzecia natomiast służy do prezentacji zmieniających się stylów. Każdy przycisk ma przypisaną procedurę obsługi zdarzenia onclick. Są to następujące funkcje JavaScript:

- ◆ odczytajStyl(); — odczytująca style przypisane warstwie div3 i zapisująca je w polu tekstowym,
- ◆ przypiszStyl(); — przypisująca warstwie div3 style znajdujące się w polu tekstowym,
- ◆ usuńStyl(); — usuwająca wszystkie style przypisane warstwie div3.

Trzecia warstwa — div3 — ma początkowo przypisany styl:

```
background-color:#ffff00;width:210px;height:50px;
```

co oznacza: żółty kolor tła, szerokość 210 pikseli i wysokość 50 pikseli.

Treść wymienionych wyżej funkcji JavaScript została przedstawiona na listingu 4.45.

Listing 4.45. Funkcje manipulujące stylami

```
<script type="text/javascript">
  function odczytajStyl()
  {
    var tf1 = document.getElementById("tf1");
    var div3 = document.getElementById("div3");
    var styl = div3.getAttribute("style");
    tf1.value = styl;
  }
  function przypiszStyl()
  {
    var tf1 = document.getElementById("tf1");
    var div3 = document.getElementById("div3");
    var styl = tf1.value;
    div3.setAttribute("style", styl);
  }
</script>
```

```
function usuńStyl()
{
    var div3 = document.getElementById("div3");
    var tf1 = document.getElementById("tf1");
    div3.removeAttribute("style");
    tf1.value = "";
}
</script>
```

Pierwsze dwa wiersze każdej z funkcji są takie same. Jest to pobranie za pomocą metody getElementById odwołań do pola tekstowego tf1 oraz warstwy div3. Funkcja odczytajStyl ma pobrać style przypisane warstwie div3 i wyświetlić je w polu tekstopowym. Do wykonania pierwszej z tych czynności używa metody getAttribute, przekazując jej w postaci argumentu ciąg style (czyli nazwę atrybutu, którego wartość ma być pobrana). A zatem po wykonaniu instrukcji:

```
styl = div3.getAttribute("style");  
zmienna styl będzie zawierała ciąg znaków opisujący styl warstwy div3. Wartość tej zmiennej jest następnie zapisywana w polu tekstopowym tf1:
```

```
tf1.value = styl;
```

Dzięki czemu pojawia się na ekranie, co jest widoczne na rysunku 4.32. Zwróćmy uwagę, że nie jest to wcale dokładnie to, co zostało zapisane w kodzie (X)HTML. Styl przypisany warstwie div3 na listingu 4.44 to:

```
background-color:#ffff00; width:210px; height:50px;
```

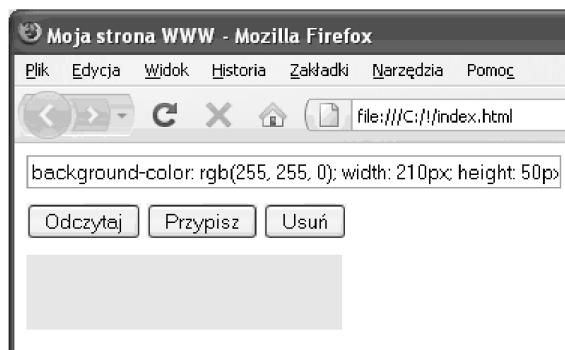
tymczasem wartość odczytana to:

```
background-color: rgb(255, 255, 0); width: 210px; height: 50px;
```

Oba zapisy są równoważne, ale nie są identyczne. Jest tak dlatego, że odczytujemy styl zinterpretowany przez przeglądarkę, a nie kod źródłowy (z podobną sytuacją mieliśmy do czynienia w lekcji 13., gdy odczytywaliśmy właściwość innerHTML).

Rysunek 4.32.

Style przypisane warstwie pojawiły się w oknie tekstopowym



Funkcja przypiszStyl wykonuje operację odwrotną do odczytajStyl — odczytuje styl zapisany w oknie tekstowym:

```
var styl = tf1.value;
```

i za pomocą metody setAttribute przypisuje go warstwie div3:

```
div3.setAttribute("style", styl);
```

Pierwszym argumentem tej metody jest nazwa atrybutu (czyli style), a drugim — wartość tego atrybutu. Oczywiście, funkcja ta zadziała właściwie tylko wtedy, gdy w polu tekstowym znajduje się ciąg znaków reprezentujący prawidłowy styl CSS.

Najprostsze zadanie wykonuje funkcja usunStyl. Korzysta z metody removeAttribute do usunięcia stylu:

```
div3.removeAttribute("style");
```

Usuwa także wartość zapisaną w polu tekstowym:

```
tf1.value = "";
```

Opisana metoda działa bez problemów w przeglądarkach, takich jak Firefox czy Opera, jednak nie w Internet Explorerze. Tu przeglądarka ta działa (jak w wielu innych przypadkach) niestandardowo. Nie oznacza to jednak, że w tym produkcie nie można przypisać dynamicznie stylu w postaci tekstowej. Po prostu właściwość style jest inaczej interpretowana. Jeśli kod z listingów 4.44 i 4.45 uruchomimy w Internet Explorerze i klikniemy przycisk *Odczytaj*, nie zobaczymy ciągu opisującego styl, ale ciąg znaków [object]. Jest to widoczne na rysunku 4.33.

Rysunek 4.33.
Niestandardowe
zachowanie
Internet Explorera



Wiemy, że odpowiada za to fragment kodu:

```
var styl = div3.getAttribute("style");
tf1.value = styl;
```

Oznacza to, że metoda getAttribute zamiast ciągu znaków opisującego styl zwróciła obiekt opisujący styl. Jest to de facto (bardzo przydatna, o czym przekonamy się w kolejnej części lekcji) właściwość style danego obiektu (X)HTML (w naszym przypadku właściwość style warstwy div3). Obiekt taki posiada właściwość o nazwie cssText, a jest ona tekstową interpretacją stylów. Jeśli więc chcemy je odczytać, zamiast napisać:

```
tf1.value = styl;
```

trzeba napisać:

```
tf1.value = styl.cssText;
```

Podczas przypisywania stylów należy natomiast modyfikować właściwość style wartości div3, zmieniając atrybut cssText:

```
div3.style.setAttribute("cssText", styl);
```

Aby więc kod z listingu 4.45 zadziałał prawidłowo w przeglądarce Internet Explorer, funkcje odczytajStyl i przypiszStyl należy zmienić na zaprezentowane na listingu 4.46.

Listing 4.46. Modyfikacja stylów w Internet Explorerze

```
function odczytajStyl()
{
    var tf1 = document.getElementById("tf1");
    var div3 = document.getElementById("div3");
    styl = div3.getAttribute("style");
    tf1.value = styl.cssText;
}
function przypiszStyl()
{
    var tf1 = document.getElementById("tf1");
    var div3 = document.getElementById("div3");
    styl = tf1.value;
    div3.style.setAttribute("cssText", styl);
}
```

Obiekt style

Opisany w poprzedniej części lekcji sposób obsługi stylów jest dobry, jeżeli chcemy odczytać wszystkie atrybuty stylu danego elementu bądź na raz je przypisać. Jeśli jednak chcemy zmodyfikować lub pobrać tylko jeden atrybut stylu CSS, np. background-color czy width, będzie to utrudnione. W przypadku odczytu trzeba by analizować cały otrzymany ciąg, a w przypadku zapisu trzeba by równocześnie zapisać wszystkie inne atrybuty.

Na szczęście, jest inny sposób dostępu do stylów. Otóż, każdy obiekt obrazujący element witryny zawiera właściwość style opisującą styl danego elementu (tak, jest to dokładnie ta właściwość, którą zwraca Internet Explorer po wywołaniu getAttribute("style")). Dokładniej mówiąc: obiekt style odzwierciedla atrybut style danego znacznika z kodu (X)HTML.

Jeżeli wskazanie do danego elementu witryny znajduje się w zmiennej obj, dostęp do obiektu style otrzymamy, pisząc:

```
obj.style;
```

W rzeczywistości obiekt ten jest kolekcją atrybutów CSS przypisanych danemu elementowi strony. Należy przy tym pamiętać, że jeśli dany atrybut jest definiowany w kodzie (X)HTML strony jako:

nazwa-trybutu

odwołanie do niego w kodzie JavaScript będzie miało postać:

nazwaAtrybutu

Przykładowo atrybut:

font-weight

jako właściwość obiektu style będzie miał postać:

fontWeight

Atrybut:

background-color

przybierze postać:

backgroundColor

Odwołania takie działają we wszystkich popularnych przeglądarkach, nie musimy więc pisać kilku wersji kodu. Wykorzystajmy to w praktyce. Napiszemy skrypt podobny do skryptu z poprzedniego przykładu, który jednak będzie pozwalał na osobną edycję każdego z atrybutów stylu przypisanego warstwie znajdującej się na witrynie. Skrypt będzie działał w każdej przeglądarce. Zaczniemy od kodu (X)HTML. Jego treść została zaprezentowana na listingu 4.47.

Listing 4.47. Strona pozwalająca na edycję poszczególnych atrybutów stylu CSS

```
<body>
    <div style="margin-bottom:10px;">
        Szerokość: <input type="text" id="tfW" size="20" />
        Wysokość: <input type="text" id="tfH" size="20" />
        Kolor tła: <input type="text" id="tfBC" size="20" />
    </div>
    <div style="margin-bottom:10px;">
        <input type="button" value="Odczytaj"
            onclick="odczytajStyl();"/>
        <input type="button" value="Przypisz"
            onclick="przypiszStyl();"/>
        <input type="button" value="Usuń"
            onclick="usuńStyl();"/>
    </div>
    <div id="div3"
        style="background-color:#ffff00;width:210px;height:50px;">
    </div>
</body>
```

Układ warstw na stronie jest taki sam jak w przykładzie z listingu 4.44. Na drugiej znajdują się przyciski służące do odczytywania, zapisywania i usuwania stylów, trzecia służy do prezentacji wyników tych operacji. Zmieniła się całkowicie treść pierwszej warstwy. Obecnie zawiera ona trzy pola tekstowe. Pierwsze odpowiada szerokości, drugie — wysokości, a trzecie — kolorowi tła warstwy div3. Strona będzie więc miała postać zaprezentowaną na rysunku 4.34 (aby nie zaciemniać kodu, z prezentowanego listingu została usunięta tabela formatująca układ pól tekstowych — jest ona jednak zawarta w plikach przykładów dołączonych do książki). Na listingu 4.48 znajduje się kod funkcji stanowiących procedury obsługi zdarzeń onclick poszczególnych przycisków.

Rysunek 4.34.

Wygląd witryny pozwalającej na manipulację stylami CSS



Listing 4.48. Funkcje obsługujące przyciski zmieniające style CSS

```
<script type="text/javascript">
    function odczytajStyl()
    {
        var tfW = document.getElementById("tfW");
        var tfH = document.getElementById("tfH");
        var tfBC = document.getElementById("tfBC");
        var div3 = document.getElementById("div3");

        tfW.value = div3.style.width;
        tfH.value = div3.style.height;
        tfBC.value = div3.style.backgroundColor;
    }
    function przypiszStyl()
    {
        var tfW = document.getElementById("tfW");
        var tfH = document.getElementById("tfH");
        var tfBC = document.getElementById("tfBC");
        var div3 = document.getElementById("div3");

        div3.style.width = tfW.value;
        div3.style.height = tfH.value;
        div3.style.backgroundColor = tfBC.value;
    }

```

```
function usuńStyl()
{
    var tfW = document.getElementById("tfw");
    var tfH = document.getElementById("tfH");
    var tfBC = document.getElementById("tfBC");
    var div3 = document.getElementById("div3");

    div3.style.width = "";
    div3.style.height = "";
    div3.style.backgroundColor = "";

    tfH.value = "";
    tfW.value = "";
    tfBC.value = "";
}
</script>
```

Początek każdej z funkcji jest taki sam, każda z nich wymaga bowiem pobrania odwołań do wszystkich pól tekstowych oraz do warstwy div3. Dane są zapisywane w zmiennych tfW (pole odpowiadające szerokości), tfH (pole odpowiadające wysokość), tfBC (pole odpowiadające kolorowi tła), div3 (warstwa div3).

Funkcja odczytajStyl odczytuje atrybuty stylu warstwy div3 i zapisuje w odpowiadających im polach tekstowych. Każdy atrybut zapisany jest jako właściwość obiektu style. A zatem szerokość (atribut width) odczytujemy, odwołując się do właściwości width, wysokość (atribut height) — odwołując się do właściwości height, kolor tła (atribut background-color) — odwołując się do właściwości backgroundColor (ponownie zwróćmy uwagę, że w przypadku atrybutów o nazwach składających się z jednego słowa nazwy odpowiadających im właściwości mają taką samą postać, natomiast dla atrybutów dwuczłonowych postać właściwości jest nieco inna). Odczytane dane są przypisywane właściwości value odpowiednich pól tekstowych, dzięki czemu pojawiają się na ekranie.

Funkcja przypiszStyl odczytuje dane z pól tekstowych i przypisuje je właściwościom odpowiadającym atrybutom width, height i background-color (właściwość background-Color). Wykonuje więc operację odwrotną do odczytajStyl. Należy zwrócić uwagę, że funkcja w żaden sposób nie bada poprawności wprowadzonych danych (np. czy prawidłowo został określony kolor tła lub czy do liczb określających wysokość i szerokość dodane zostało określenie miary). Część przypisań może więc nie zostać uwzględniona.

Ostatnia z funkcji — usuńStyl — ma usunąć atrybuty stylu CSS warstwy div3. Robi to, po prostu przypisując odpowiednim właściwościom obiektu style puste ciągi znaków. W ten sposób wartości atrybutów width, height i backgroundColor zostaną „wyzerowane”.

Wykonajmy jeszcze jeden przykład, w którym modyfikowana będzie konkretna właściwość obiektu style, czyli jeden wybrany atrybut stylu CSS. W niekonwencjonalny sposób wykorzystamy przy tym przyciski. Spójrzmy na kod zaprezentowany na listingu 4.49.

Listing 4.49. Dynamiczna zmiana stylów

```
<body>
  <div style="margin-bottom:10px;">
    <input type="button" value="Zmień kolor tła"
      onmouseover="zmieńKolorTła('#ff00ff');"
      onmouseout="zmieńKolorTła('#ee0000');"/>
    <input type="button" value="Zmień kolor tekstu"
      onmouseover="zmieńKolorTekstu('#ffffff');"
      onmouseout="zmieńKolorTekstu('#000000');"/>
  </div>
  <div id="div3"
    style="background-color:#ee0000;width:210px;height:50px;">
    <p id="pt1">Ten tekst może zmienić kolor.</p>
  </div>
</body>
```

Strona zawiera dwie warstwy. Na pierwszej znajdują się dwa przyciski, na drugiej — akapit tekstowy. Druga warstwa ma przypisany styl określający jej kolor tła oraz wysokość i szerokość, akapitowi został natomiast nadany identyfikator id1. Odmienność niż w większości dotychczasowych przykładów wygląda natomiast definicja przycisków. Żaden z nich nie obsługuje zdarzenia onclick, a więc żaden nie będzie reagował na kliknięcia. Będą natomiast reagować na ruchy kurSORA myszy. Odpowiadają za to zdarzenia onmouseover i onmouseout. W przypadku pierwszego przycisku oba zdarzenia otrzymały procedurę obsługi w postaci funkcji zmieńKolorTła, a w przypadku drugiego — zmieńKolorTekstu. To oznacza, że:

- ◆ kiedy kurSOR najedzie na pierwszy przycisk, zostanie wywołana metoda zmieńKolorTła z argumentem #ff00ff;
- ◆ kiedy kurSOR opuści obszar pierwszego przycisku, zostanie wywołana metoda zmieńKolorTła z argumentem #ee0000;
- ◆ kiedy kurSOR najedzie na drugi przycisk, zostanie wywołana metoda zmieńKolorTekstu z argumentem #ffffff;
- ◆ kiedy kurSOR opuści obszar drugiego przycisku, zostanie wywołana metoda zmieńKolorTekstu z argumentem #000000;

Dzięki temu kolor tła warstwy oraz znajdującego się na niej akapitu tekowego będą się zmieniały, w zależności od tego, czy nad przyciskiem znajduje się kurSOR, a jeśli tak, to nad którym.

Napiszmy zatem treść funkcji zmieńKolorTła i zmieńKolorTekstu, tak by strona zachowywała się w opisany sposób. Treść tych funkcji znajduje się na listingu 4.50.

Listing 4.50. Funkcje zmieniające kolor tekstu i tła

```
<script type="text/javascript">
  function zmieńKolorTła(kolor)
  {
    var div3 = document.getElementById("div3");
    div3.style.backgroundColor = kolor;
  }
```

```
function zmieńKolorTekstu(kolor)
{
    var pt1 = document.getElementById("pt1");
    pt1.style.color = kolor;
}
</script>
```

Kod JavaScript jest tu wyjątkowo prosty. Funkcja zmieńKolorTła ma zmienić właściwość backgroundColor obiektu style warstwy div3 na wartość przekazaną w postaci argumentu. Pobiera więc odwołanie do warstwy i dokonuje przypisania, dokładnie na takiej samej zasadzie jak we wcześniej prezentowanych przykładach. Funkcja zmieńKolorTekstu działa bardzo podobnie. Ma zmienić właściwość color (kolor pierwszo-planowy, w tym przypadku — kolor tekstu) obiektu style akapitu pt1 na wartość przekazaną w postaci argumentu. Kilka prostych instrukcji pozwala na pełną obsługę witryny.

Zauważmy też, że obie funkcje są do siebie bardzo podobne. Może więc warto połączyć je w jedną? Przemyślenie, czy faktycznie warto, a jeśli tak, to jak powinien wtedy wyglądać kod, będzie dobrym ćwiczeniem do samodzielnego wykonania.

Właściwość className

Dotychczas modyfikowaliśmy konkretne atrybuty stylu przypisanego wybranemu elementowi witryny. To nie jedyna możliwość wpływania na prezentację składowych strony. Stosując JavaScript, możemy również modyfikować wartość atrybutu class danego znacznika (X)HTML, a tym samym dokonywać przełączenia między różnymi stylami. Atrybut class jest odzwierciedlany jako właściwość className obiektu odpowiadającego danemu znacznikowi. Jeśli np. w kodzie (X)HTML istnieje akapit zdefiniowany jako:

```
<p id="pt1" class="prostyakapit">Tekst akapitu</p>
```

po pobraniu odwołania do niego za pomocą metody getElementById:

```
pt1 = document.getElementById("pt1");
```

zmianę wartości atrybutu class osiągniemy, pisząc:

```
pt.className = "wartość";
```

Sprawdźmy to w praktyce. Przygotujemy dwa style i będziemy dokonywać przełączenia między nimi. Definicja stylów będzie standardowa — użyjemy znacznika <style>. Oba zostały przedstawione na listingu 4.51.

Listing 4.51. Zestaw prostych stylów

```
<style type="text/css">
.warstwaZwykła
{
    background-color:#EFEFEF;
    border:1px dotted #000000;
    width:300px;
    height:100px;
}
```

```
.warstwaWyróżniona
{
    background-color:#AFAFAF;
    border:2px solid #ff0000;
    width:300px;
    height:100px;
}
</style>
```

Style te określają kolor tła (background-color), szerokość (width), wysokość (height) oraz wygląd obramowania (border) wybranego elementu strony. Definicja obramowania składa się z trzech części: szerokości, typu i koloru. Tak więc zapis:

1px dotted #000000;

oznacza kropkowane (ang. *dotted*) obramowanie o szerokości 1 piksela w kolorze czarnym, a:

2px solid #ff0000;

to obramowanie o szerokości 2 pikseli, ale jednolite (ang. *solid*), w kolorze czerwonym. Całą treść z listingu 4.51 należy umieścić w sekcji <head> witryny.

Style będziemy przypisywać umieszczonej na stronie warstwie. Zmiany będą dokonywane za pomocą przycisków. Wszystkie wymienione elementy musimy więc umieścić w kodzie (X)HTML, który został zaprezentowany na listingu 5.52.

Listing 4.52. Witryna z przyciskami zmieniającymi style

```
<body onload="zmieńTypWarstwy('warstwaZwykła');">
    <div style="margin-bottom:10px;">
        <input type="button" value="Warstwa zwykła"
               onclick="zmieńTypWarstwy('warstwaZwykła');"
        />
        <input type="button" value="Warstwa wyróżniona"
               onclick="zmieńTypWarstwy('warstwaWyróżniona');"
        />
    </div>
    <div id="div2">
        Styl tej warstwy będzie się zmieniał.
    </div>
</body>
```

Na stronie znajdują się dwie warstwy. Pierwsza zawiera przyciski pozwalające na zmianę stylu przypisanego drugiej. Oba przyciskom została przypisana procedura obsługi zdarzenia onclick w postaci funkcji zmieńTypWarstwy. Funkcja ta została również przypisana zdarzeniu onload sekcji <body>. Dzięki temu po załadowaniu strony warstwie div2 zostanie przypisany konkretny styl, określany przez argument funkcji zmieńTypWarstwy. Argumentem jest ciąg znaków ustalający nazwę stylu. Treść funkcji zmieńTypWarstwy została przedstawiona na listingu 4.53.

Listing 4.53. Funkcja zmieniająca styl warstwy

```
<script type="text/javascript">
    function zmieńTypWarstwy(typ)
    {
        var div2 = document.getElementById("div2");
        div2.className = typ;
    }
</script>
```

Kod jest tak prosty, że w zasadzie nie wymaga tłumaczenia. Skoro bowiem funkcja jako argument otrzymuje nazwę stylu, który ma być przypisany warstwie div2, wystarczy pobrać odwołanie do tej warstwy i przypisać wartość argumentu właściwości className. Tak właśnie dzieje się w powyższym przykładzie.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 16.1.

Zmodyfikuj kod przykładu z listingu 4.45, tak by operacja usunięcia stylu wymagała potwierdzenia.

Ćwiczenie 16.2.

Umieść na stronie przycisk. Najechanie na niego kursem myszy powinno spowodować zmianę koloru przycisku i koloru znajdującego się na nim tekstu.

Ćwiczenie 16.3.

Bazując na przykładzie z listingu 4.47 i 4.48, napisz skrypt umożliwiający modyfikację wysokości i szerokości umieszczonej na stronie warstwy. Użytkownik ma mieć możliwość określenia za pomocą pól wyboru jednostki miary (piksele lub centymetry).

Ćwiczenie 16.4.

Zmodyfikuj kod z listingu 4.49 i 4.50, tak aby do zmian kolorów była używana tylko jedna funkcja.

Rozdział 5.

Przetwarzanie danych

Lekcja 17. Operacje na ciągach znaków

W lekcji 2., w rozdziale 2. poznaliśmy typ łańcuchowy, czyli typ string. Dane tego typu to po prostu ciągi znaków — korzystaliśmy z nich wielokrotnie na łamach książki. W praktyce programistycznej bardzo często spotkamy się jednak z koniecznością najrozmaitszego przetwarzania ciągów znaków. Czasem trzeba będzie sprawdzić, czy w danym ciągu znajduje się jakiś podciąg, czasem wyodrębnić fragment tekstu, niekiedy zbadać, jaka jest pierwsza lub ostatnia litera. Takiej właśnie tematyce poświęcona jest lekcja 17.

Jak sprawdzić długość tekstu?

W lekcji 9., w rozdziale 3. dowiedzieliśmy się, że dane typów prostych (liczbowych, łańcuchowych itd.) w JavaScripte traktowane są jak obiekty. Dokładnie tak samo jest z ciągami znaków. Jeśli gdzieś w kodzie skryptu wystąpi ciąg znaków, spowoduje to powstanie obiektu typu string. Taki obiekt ma własne właściwości i metody, z których możemy korzystać do woli. Właściwością charakterystyczną dla obiektu typu string jest `length`¹. Zawiera ona długość ciągu, czyli liczbę zawartych w nim znaków — niewątpliwie jest więc bardzo użyteczna. Jeśli zatem w zmiennej `str` znajduje się ciąg znaków, jego długość uzyskamy, pisząc:

```
str.length
```

Oto przykład:

```
var str = "To jest przykładowy tekst.";
var długość = str.length;
alert("Długość tekstu to " + długość + " znaków.");
```

¹ Oprócz `length`, obiekt typu string, tak jak i każdy inny, zawiera również właściwości `constructor` i `prototype`. Były one omawiane w lekcji 9., w rozdziale 3.

Skoro jednak ciąg znaków w trakcie przetwarzania przez aparat wykonawczy JavaScript jest zamieniany na obiekt, odwołanie do jego właściwości (oraz metod) może być przeprowadzone bezpośrednio. Prawidłowy jest też zapis:

"ciąg znaków".length

Przykładowo:

```
var długość = "To jest przykładowy tekst.".length;
alert("Długość tekstu to " + długość + " znaków.");
```

Napiszmy zatem skrypt, który umożliwi użytkownikowi wprowadzenie tekstu i poda jego długość. Kod (X)HTML został zaprezentowany na listingu 5.1.

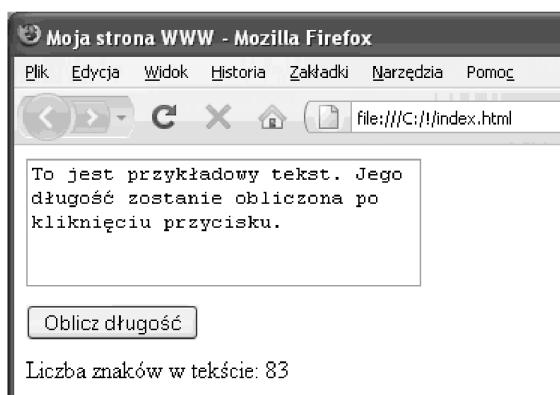
Listing 5.1. Strona podająca długość wprowadzonego tekstu

```
<body>
<div style="margin-bottom:10px;">
    <textarea id="ta1" cols="30" rows="4"></textarea>
</div><div style="margin-bottom:10px;">
    <input type="button" value="Oblicz długość"
        id="btn1"
        onclick="obliczDługość();"/>
</div>
<div id="div3"></div>
</body>
```

Mamy tu trzy warstwy. Pierwsza zawiera rozszerzone pole tekstowe, druga — przycisk, a trzecia (o identyfikatorze `div3`) służy do prezentacji wyników. Pole testowe zostało zdefiniowane za pomocą znacznika `<textarea>` i zostało mu nadany identyfikator `ta1`. Przyciskowi została przypisana procedura obsługi zdarzenia `onclick` w postaci funkcji `obliczDługość`. A zatem po wprowadzeniu tekstu do pola będzie można kliknąć przycisk *Oblicz długość* i wtedy na ekranie pojawi się informacja o liczbie znaków, co zostało zaprezentowane na rysunku 5.1. Treść funkcji `obliczDługość` znajduje się na listingu 5.2.

Rysunek 5.1.

Po kliknięciu przycisku została wyświetlona długość tekstu



Listing 5.2. Funkcja obliczająca długość tekstu

```
<script type="text/javascript">
    function obliczDługość()
    {
        var ta1 = document.getElementById("ta1");
        var div3 = document.getElementById("div3");
        var długość = ta1.value.length;
        div3.innerHTML = "Liczba znaków w tekście: " + długość;
    }
</script>
```

Funkcja za pomocą metody getElementById pobiera odwołania do pola tekstowego oraz warstwy div3 i zapisuje je w zmiennych ta1 i div3. Następnie odczytuje długość tekstu. Robi to, odwołując się do właściwości length. Skoro bowiem ciąg znaków znajdujący się polu ta1 znajduje się we właściwości value, jego długość odczytamy przy użyciu odwołania:

```
ta1.value.length
```

Liczba znaków znajdujących się we wprowadzonym tekście (po odczytaniu) jest wyświetlana na warstwie div3:

```
div3.innerHTML = "Liczba znaków w tekście: " + długość;
```

Powyższy przykład można w łatwy sposób zmodyfikować tak, by użytkownik na bieżąco był informowany o tym, ile znaków wprowadził do pola tekstowego. Gdy zajrzymy do tabeli 4.11 z lekcji 14., znajdziemy kilka zdarzeń związanych z obsługą klawiatury. Są to onkeydown, onkeypress i onkeyup. Najodpowiedniejsze do naszego celu wydaje się być onkeyup, powstaje bowiem, gdy zostanie puszczyony (rzec jasna po uprzednim wcisnięciu) dowolny klawisz. Wystarczy wtedy sprawdzić liczbę znaków i wyświetlić tę informację na ekranie. Nie musimy przy tym wcale modyfikować skryptu z listingu 5.2. Wystarczą zmiany w kodzie (X)HTML z listingu 5.1. Treść sekcji body będzie teraz wyglądała tak, jak na listingu 5.3.

Listing 5.3. Strona obliczająca liczbę znaków na bieżąco

```
<body>
    <div style="margin-bottom:10px;">
        <textarea id="ta1" cols="30" rows="4"
            onkeyup="obliczDługość();">
        </textarea>
    </div>
    <div id="div3"></div>
</body>
```

Z kodu została usunięta warstwa zawierająca przycisk — nie jest w tym przykładzie do niczego potrzebny. Zdarzeniu onkeyup pola tekstowego została natomiast przypisana procedura obsługi w postaci funkcji obliczDługość. Jest to ta sama funkcja, której treść widnieje na listingu 5.2. Tym razem będzie po prostu wykonywana po każdym puszczeniu klawisza. A więc po każdym takim zdarzeniu na ekranie pojawi się informacja o liczbie znaków w tekście znajdującym się w polu tekstowym. Oznacza to, że wyświetlana informacja będzie aktualniana na bieżąco w trakcie pisania na klawiaturze.

Metody formatujące ciągi znaków

Metody dostępne w obiektach typu `String` możemy podzielić na takie, które formatują ciągi znaków, oraz takie, które w jakiś sposób ciągi przetwarzają lub udostępniają różne informacje. Zajmiemy się najpierw pierwszą grupą metod. Co prawda, obecnie straciły na znaczeniu (metody te dostępne są od pierwotnych wersji JavaScriptu), ale warto wiedzieć, że istnieją i jak ich używać. Metody formatujące zostały zebrane w tabeli 5.1.

Użyjmy kilku metod przedstawionych w tabeli metod. Napiszemy kod witryny zawierającej cztery pola wyboru typu radio, pole tekstowe oraz przycisk. Za pomocą pól wyboru będzie można wybrać efekt, który zostanie zastosowany w stosunku do tekstu wprowadzonego w polu tekstowym. Witryna będzie więc wyglądała tak, jak na rysunku 5.2. Kod (X)HTML takiej strony został zaprezentowany na listingu 5.4.

Kod witryny składa się z trzech warstw. Pierwsza zawiera cztery pola wyboru zdefiniowane za pomocą znaczników `<input>` z parametrem `type="radio"` ustawionym na `radio`. Właściwość `name` wszystkich pól została ustawiona na `grp1`, co sprawiło, że stanowią one jedną grupę i na raz będzie mogło być zaznaczone tylko jedno z nich (są polami wzajemnie się wykluczającymi). Każde pole ma również przypisany identyfikator określający, jaki efekt włącza: `rbBlink` (znacznik `<blink>`, tekst migający, metoda `blink`), `rbBold` (znacznik ``, tekst pogrubiony, metoda `bold`), `rbItalics` (znacznik `<i>`, tekst pochylony, metoda `italics`), `rbStrike` (znacznik `<strike>`, tekst przekreślony, metoda `strike`).

Druga warstwa zawiera pole tekstowe oraz przycisk. Polu został nadany identyfikator `tf1`, za którego pomocą będzie się można odwoać do tego elementu w skrypcie i odczytać wprowadzony tam tekst. Zdarzeniu `onClick` przycisku została natomiast przypisana procedura obsługi w postaci funkcji `przetwórzTekst`. Trzecia warstwa ma identyfikator `div3` i początkowo jest pusta. Będzie się na niej pojawił tekst wprowadzony do pola tekstowego i przetworzony przez wybraną w polach wyboru metodę.

Treść funkcji `przetwórzTekst` została przedstawiona na listingu 5.5.

Najpierw odczytywane są i zapisywane w zmiennych `tf1` i `div3` odwołania pola tekstowego `tf1` i warstwy `div3`, a tekst zapisany w polu `tf1` jest przypisywany zmiennej `tekst`. Następnie w złożonej instrukcji warunkowej `if...else if` badane jest, które z pól wyboru jest zaznaczone, czyli ma właściwość `checked` ustawioną na `true`. W zależności od tego, wywoływana jest jedna z metod obiektu `tekst`, czyli `blink`, `bold`, `italice` lub `strike`, a efekt jej wywołania jest przypisywany z powrotem zmiennej `tekst`. Tym samym, jeżeli jedno z pól wyboru jest zaznaczone, tekst z pola tekstowego zostanie przetworzony przez odpowiadającą polu metodę. Jeżeli zaś żadne z pól nie jest zaznaczone (w praktyce taka sytuacja nie powinna mieć miejsca), tekst pozostanie niezmieniony.

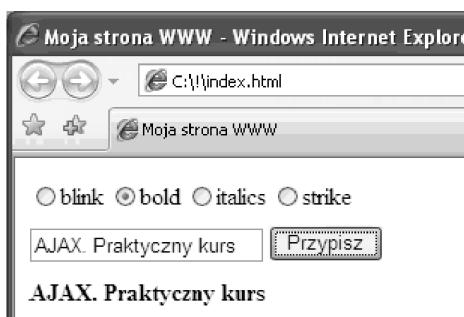
Na końcu kodu wartość zmiennej `tekst`, niezależnie od tego, czy została przetworzona, czy też nie, jest przypisywana właściwości `innerHTML` warstwy `div3`, dzięki czemu zostaje wyświetlona na ekranie.

Tabela 5.1. Metody formatujące ciągi

Metoda	Wywołanie	Opis	Przykład wywołania	Efekt wywołania
anchor	<code>str.anchor ↳(param)</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <a>串 z parametrem name równym param.	"abc".anchor ↳("def")	abc
big	<code>str.big()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <big>.	"abc".big ↳("def")	<big>abc</big>
blink	<code>str.blink()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <blink>.	"abc".blink ↳("def")	<blink>abc ↳</blink>
bold	<code>str.bold()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik .	"abc".bold ↳("def")	abc
fixed	<code>str.fixed()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <tt>.	"abc".fixed ↳("def")	<tt>abc</tt>
fontcolor	<code>str.fontcolor ↳(kolor)</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik , z parametrem color równym argumentowi kolor.	"abc".fontcolor ↳("red")	abc ↳
fontsize	<code>str.fontsize ↳(rozmiar)</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik , z parametrem size równym argumentowi rozmiar.	"abc". ↳fontsize(7)	abc
italics	<code>str. ↳italics()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <i>.	"abc".italics()	<i>abc</i>
link	<code>str.link ↳(param)</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <a> z parametrem href równym param.	"abc".link ↳("http://nazwa. ↳domeny")	abc
small	<code>str.small()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <small>.	"abc".small()	<small>abc ↳</small>
strike	<code>str.strike()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <strike>.	"abc".strike()	<strike>abc ↳</strike>
sub	<code>str.sub()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <sub>.	"abc".sub()	_{abc}
sup	<code>str.sup()</code>	Zwraca串 znaków, w którym串 reprezentowany przez str串 został ujęty w znacznik <sup>.	"abc".sup()	^{abc}

Rysunek 5.2.

Pola wyboru pozwalają na wybór rodzaju tekstu

**Listing 5.4.** Strona zawierająca pola wyboru, pole tekstowe i przycisk

```
<body>
<div style="margin-bottom:10px;">
    <input type="radio" id="rbBlink" name="grp1">blink
    <input type="radio" id="rbBold" name="grp1">bold
    <input type="radio" id="rbItalics" name="grp1">italics
    <input type="radio" id="rbStrike" name="grp1">strike
</div>
<div style="margin-bottom:10px;">
    <input type="text" id="tf1">
    <input type="button" id="btn1"
        value="Przypisz" onclick="przetwórzTekst()"/>
</div>
<div id="div3"></div>
</body>
```

Listing 5.5. Funkcja przetwarzająca tekst z pola tekstowego

```
<script type="text/javascript">
function przetwórzTekst()
{
    var tf1 = document.getElementById("tf1");
    var div3 = document.getElementById("div3");
    tekst = tf1.value;
    if(document.getElementById("rbBlink").checked)
        tekst = tekst.blink();
    else if(document.getElementById("rbBold").checked)
        tekst = tekst.bold();
    else if(document.getElementById("rbItalics").checked)
        tekst = tekst.italics();
    else if(document.getElementById("rbStrike").checked)
        tekst = tekst.strike();
    div3.innerHTML = tekst;
}
```

Przetwarzanie ciągów

Działanie metod przedstawionych w poprzedniej części lekcji można w bardzo prosty sposób zasymulować ręcznie, wprowadzając instrukcje łączące ciągi. I tak występujące na listingu 5.5 wywołanie:

```
var tekst = tekst.bold();
```

można z powodzeniem zastąpić przez:

```
var tekst = "<b>" + tekst + "</b>";
```

Efekt działania tych konstrukcji będzie identyczny.

Efektów działania omawianych w tej części lekcji metod przetwarzających ciągi nie da się już tak łatwo zasymulować, a wykonują one bardzo wiele zadań wyjątkowo przydatnych przy pracy z ciągami znaków. Metody te zostały zebrane w tabeli 5.2.

Tabela 5.2. Metody przetwarzające ciągi

Metoda	Wywołanie	Opis
charAt	<code>str.charAt(index)</code>	Zwraca znak ciągu znajdujący się pod indeksem <i>indeks</i> .
charCodeAt	<code>str.charCodeAt(index)</code>	Zwraca kod znaku znajdującego się pod indeksem <i>indeks</i> . W wersji JavaScript 1.2 jest to kod ISO-Latin1, a od wersji 1.3 (oraz w JScript) — kod Unicode.
concat	<code>str.concat(str2, str3, ..., strN)</code>	Łączy łańcuchy znakowe. Wynikiem działania metody jest ciąg powstały po połączeniu łańcucha <i>str</i> i wszystkich łańcuchów przekazanych w postaci argumentów.
fromCharCode	<code>String.fromCharCode(↳(kod1, kod2, ..., ↳kodN))</code>	Zwraca ciąg znaków, których kody zostały przekazane w postaci argumentów. Jest to metoda statyczna, do jej wykorzystania nie jest zatem konieczne tworzenie obiektu typu <i>String</i> .
indexOf	<code>str.indexOf(value ↳[, index])</code>	Zwraca <i>indeks</i> wystąpienia ciągu <i>value</i> w ciągu <i>str</i> . Jeżeli podany zostanie opcjonalny drugi argument, przeszukiwanie ciągu <i>str</i> rozpocznie się od znaku znajdującego się pod indeksem <i>index</i> .
lastIndexOf	<code>str.lastIndexOf(value ↳[, index])</code>	Działa podobnie do metody <i>indexOf</i> , z tą różnicą, że ciąg <i>str</i> przeszukiwany jest od końca.
match	<code>str.match(wyrReg)</code>	Zwraca część ciągu <i>str</i> pasującego do wyrażenia regularnego <i>wyrReg</i> . Jeżeli żadna część ciągu nie pasuje do wyrażenia, zwraca wartość <i>null</i> .
replace	<code>str.replace(↳(wyrReg, tekst))</code>	Zwraca ciąg, w którym podciagi opisane przez wyrażenie regularne <i>wyrReg</i> zostały zamienione na ciąg znaków reprezentowany przez <i>tekst</i> .
search	<code>str.search(wyrReg)</code>	Sprawdza, czy ciąg opisany przez wyrażenie regularne <i>wyrReg</i> występuje w ciągu <i>str</i> . Jeśli tak, zwracany jest indeks wystąpienia, jeśli nie, zwracana jest wartość -1.

Tabela 5.2. Metody przetwarzające ciągi (ciąg dalszy)

Metoda	Wywołanie	Opis
slice	<code>str.slice(start[, end])</code>	Zwraca podciąg ciągu <i>str</i> rozpoczynający się od indeksu wskazywanego przez <i>start</i> i kończący się wraz z końcem ciągu <i>str</i> , lub też, jeśli został podany parametr <i>end</i> , w indeksie wskazywanym przez ten parametr.
split	<code>str.split([separator ↳[, limit]])</code>	Dzieli ciąg znaków <i>str</i> na podciągi względem parametru <i>separator</i> i zwraca je w postaci tablicy. Parametr <i>separator</i> może być ciągiem znaków lub wyrażeniem regularnym, jeżeli nie zostanie podany, zwrócony zostanie pełny ciąg znaków. Parametr <i>limit</i> jest wartością całkowitą wskazującą maksymalną liczbę elementów tablicy wynikowej.
substr	<code>str.substr(index ↳[, ile])</code>	Zwraca podciąg ciągu <i>str</i> rozpoczynający się od znaku o indeksie <i>indeks</i> , zawierający liczbę znaków wskazywanych przez parametr <i>ile</i> . Jeżeli parametr <i>ile</i> nie zostanie podany, zwracane są wszystkie znaki od indeksu <i>indeks</i> do końca ciągu <i>str</i> .
substring	<code>str.substring ↳(indeks1, indeks2)</code>	Zwraca podciąg ciągu <i>str</i> rozpoczynający się od znaku o indeksie <i>indeks1</i> i kończący się przed znakiem o indeksie <i>indeks2</i> .
toLowerCase	<code>str.toLowerCase()</code>	Zwraca ciąg, w którym wszystkie znaki ciągu <i>str</i> zostały zamienione na małe litery. Wywołanie metody nie wpływa na zawartość ciągu <i>str</i> .
toUpperCase	<code>str.toUpperCase()</code>	Zwraca ciąg, w którym wszystkie znaki ciągu <i>str</i> zostały zamienione na duże litery. Wywołanie metody nie wpływa na zawartość ciągu <i>str</i> .

Spójrzmy na przykłady użycia tych metod.

Działanie metody `charAt` wydaje się oczywiste. Skoro zwraca znak znajdujący się pod wskazanym indeksem, to przykładowe wywołanie:

```
var c = "łaka".charAt(2);
```

spowoduje przypisanie zmiennej *c* znaku *k*. Znak ten znajduje się bowiem pod indeksem 2 (indeksowanie ciągu zaczyna się od 0). Należy jedynie pamiętać, że sekwencje znaków specjalnych tworzą w rzeczywistości jeden znak, czyli w wyniku wykonania instrukcji:

```
var c1 = "\\\t".charAt(0);
var c2 = "\\\t".charAt(1);
```

otrzymamy przypisanie zmiennej *c1* znaku \, a zmiennej *c2* — znaku tabulacji poziomej.

Metoda `charCodeAt` powoduje pobranie kodu znaku znajdującego się pod wskazanym indeksem. Wykonanie przykładowych instrukcji:

```
var c1 = "łaka".charCodeAt(0);
var c2 = "łaka".charCodeAt(2);
```

spowoduje przypisanie zmiennej c1 wartości 322 (kod znaku *l* w postaci dziesiętnej), a zmiennej c2 — wartości 107 (kod znaku *k* w postaci dziesiętnej). Oczywiście, dla sekwencji znaków specjalnych obowiązują te same zasady, co dla metody charAt, czyli wywołanie:

```
var c3 = "\\\t".charCodeAt(0);
```

spowoduje przypisanie zmiennej c3 wartości 92 (kod znaku \).

Metoda concat zwraca串被当作连接两个或多个字符串的参数。 A zatem przykładowe wywołanie:

```
var s1 = "abc".concat("123", "def");
```

spowoduje przypisanie zmiennej s1串abc123def.

Metoda fromCharCode jest nieco inna od dotychczas opisanych. Można ją wywołać, nie tworząc żadnego obiektu typu String (takie metody nazywa się statycznymi). W wyniku jej działania powstaje串被当作连接两个或多个字符串的参数。 A zatem przykładowe wywołanie:

```
var s1 = String.fromCharCode(322, 261, 107, 97);
```

spowoduje przypisanie zmiennej s1串ąka.

Metoda indexOf pozwala stwierdzić, czy w串被当作连接两个或多个字符串的参数。 A zatem sprawdza, czy w串被当作连接两个或多个字符串的参数。 Jeżeli istnieje, zwracany jest indeks wystąpienia, jeśli nie — wartość -1. To oznacza, że instrukcja:

```
var indeks = "piękna łąka".indexOf("na");
```

spowoduje przypisanie zmiennej indeks wartości 4 — bowiem串被当作连接两个或多个字符串的参数。 Z kolei instrukcja:

```
var indeks = "piękna łąka".indexOf("one");
```

spowoduje przypisanie zmiennej indeks wartości -1, bowiem w串被当作连接两个或多个字符串的参数。

Omawiana metoda umożliwia użycie drugiego argumentu. Określa on indeks znaku, od którego ma się zacząć przeszukiwanie串被当作连接两个或多个字符串的参数。 Znaczy to, że przykładowe wywołanie:

```
var indeks = "piękna łąka".indexOf("ą", 4);
```

spowoduje przypisanie zmiennej indeks wartości 8 — bowiem串被当作连接两个或多个字符串的参数。 Natomiast wywołanie:

```
var indeks = "piękna łąka".indexOf("ą", 9);
```

spowoduje przypisanie zmiennej indeks wartości -1 — bowiem串被当作连接两个或多个字符串的参数。 A zatem przykładowe wywołanie:

Metoda `lastIndexOf` działa na tej samej zasadzie, co `indexOf`, ale przeszukuje ciąg od końca. Jeśli więc wykonamy serię instrukcji:

```
var i1 = "błękitne niebo".lastIndexOf("ne");
var i2 = "błękitne niebo".lastIndexOf("na");
var i3 = "błękitne niebo".lastIndexOf("ki", 6);
var i4 = "błękitne niebo".lastIndexOf("ki", 2);
```

okaże się, że:

- ◆ zmienna `i1` zawiera wartość 6, bowiem ciąg ne rozpoczyna się w indeksie 6;
- ◆ zmienna `i2` zawiera wartość -1, bowiem ciąg na nie występuje w ciągu błękitne niebo;
- ◆ zmienna `i3` zawiera wartość 3, bowiem przeszukiwanie rozpoczyna się w indeksie 6 (licząc od początku), a ciąg ki rozpoczyna się w indeksie 3;
- ◆ zmienna `i4` zawiera wartość -1, bowiem ciąg ki rozpoczyna się w indeksie 3, a przeszukiwanie rozpoczyna się w indeksie 2 (i dąży do indeksu 0);

Metoda `match` pozwala stwierdzić, czy w danym ciągu występuje podciąg odpowiadający wyrażeniu regularnemu przekazanemu w postaci argumentu. Wyrażeniami regularnymi zajmiemy się dopiero w lekcji 19. Funkcja pozwala również na pracę ze zwykłymi ciągami, a zatem możemy stwierdzić, czy w danym ciągu występuje pewien podciąg. Jeśli tak jest, metoda zwróci ten podciąg, jeśli nie — zwróci wartość `null`. Znaczy to, że po wykonaniu instrukcji:

```
var str = "błękitne niebo".match("nie");
```

zmienna `str` będzie zawierała ciąg `nie` (bowiem ciąg `nie` jest podciągiem ciągu `błękitne niebo`), a po wykonaniu instrukcji:

```
var str = "błękitne niebo".match("abc");
```

zmienna `str` będzie zawierała wartość `null` (bowiem ciąg `abc` nie jest podciągiem ciągu `błękitne niebo`).

Metoda `replace` zamienia wszystkie podciągi zgodne z wyrażeniem regularnym (lekcja 19.) podanym jako pierwszy argument na ciąg przekazany jako drugi argument. Jeśli nie chcemy korzystać z zaawansowanych wyrażeń, ale mamy zamiar zamieniać zwykłe ciągi znaków, możemy ciąg poszukiwany umieścić między znakami `/`. Przykładowo po wykonaniu instrukcji:

```
var str = "Cześć %IMIE%. Miło Cię spotkać.".replace(/%IMIE%/, "Adam");
```

zmienna `str` będzie zawierała ciąg znaków:

Cześć Adam. Miło Cię spotkać.

bowiem ciąg `%IMIE%` zostanie zamieniony na ciąg `Adam`. Uwaga: w ten sposób zostanie zamienione tylko pierwsze wystąpienie szukanego ciągu. Jeśli chcemy zamienić wszystkie wystąpienia, należy skorzystać z opcji `g`, np. po wykonaniu instrukcji:

```
var str1 = "Ala ma kota. Ala ma też psa.".replace(/Ala/, "Ola");
var str2 = "Ala ma kota. Ala ma też psa.".replace(/Ala/g, "Ola");
```

zmienna str1 będzie zawierała ciąg znaków:

Ola ma kota. Ala ma też psa.

a zmienna str2 ciąg:

Ola ma kota. Ola ma też psa.

Drugi argument metody może zawierać sekwencje znaków specjalnych (dostępne w JavaScript od wersji 1.2 i JScript od wersji 5.5) przedstawionych w tabeli 5.3.

Tabela 5.3. Sekwencje znaków specjalnych dla metody replace

Sekwencja	Znaczenie
\$\$	znak \$
\$\$	podciąg pasujący do wyrażenia <i>wyrReg</i>
\$`	podciąg znajdujący się przed ciągiem \$\$
\$'	podciąg znajdujący się za ciągiem \$\$
\$n lub \$nn	n-ty podciąg wyrażenia <i>wyrReg</i>

Metoda search działa tak samo jak indexOf, z tą różnicą, że argumentem jest wyrażenie regularne. Odszukuje więc pierwszy podciąg pasujący do tego wyrażenia i zwraca indeks jego wystąpienia. Jeżeli w ciągu nie istnieje żaden podciąg pasujący do wyrażenia, zwracana jest wartość -1. Przykładowe wywołania:

```
var i1 = "Ten serwer ma 2GB RAM.".search(/ser[vw]er/);
var i2 = "This server has 2GB RAM.".search(/ser[vw]er/);
```

spowodują przypisanie zmiennej i1 wartości 4, a zmiennej i2 — wartości 5. Wyrażenie /ser[vw]er/ pasuje bowiem zarówno do ciągu serwer (a występuje on w ciągu głównym na 4. pozycji), jak i server (a występuje on w ciągu głównym na 5. pozycji).

Metoda slice pozwala na pobranie fragmentu danego ciągu. Jeżeli zostanie wywołana z jednym argumentem, zwrócony podciąg będzie się rozpoczynał w indeksie wskazanym przez ten argument i kończył w ostatnim znaku ciągu głównego. Jeżeli zostaną podane dwa argumenty, zwrócony zostanie podciąg przez nie wyznaczany. Znaczy to, że wywołanie:

```
var str1 = "świt w świecie świerszcza".slice(7);
```

spowoduje przypisanie zmiennej str1 wartości świecie świerszcza, a wywołanie:

```
var str2 = "świt w świecie świerszcza".slice(8, 14);
```

spowoduje przypisanie zmiennej str2 wartości wiecie.

Mozliwe jest również stosowanie argumentów o wartościach ujemnych. Wtedy są one interpretowane jako przesunięcia względem końca ciągu głównego. Przykładowo wywołanie:

```
str = "niebieska waza".slice(5, -5);
```

spowoduje przypisanie zmiennej str ciągu eska. Jest to ciąg rozpoczynający się w znaku o indeksie 5, licząc od początku, i kończący się w znaku o indeksie 5, licząc od końca. Identyczny efekt da wywołanie:

```
str = "niebieska waza".slice(-9, -5);
```

Będzie to powiem ciąg rozpoczynający się w znaku o indeksie 9, licząc od końca, i kończący się w znaku o indeksie 5, również licząc od końca.

Metoda split umożliwia podzielenie ciągu względem znaków separatora przekazanego jako pierwszy argument. Podzielony ciąg jest zwracany w postaci tablicy. Opcjonalny drugi argument pozwala określić maksymalną liczbę ciągów wynikowych (a tym samym, rozmiar tablicy wynikowej). Separator może być określony jako pojedynczy znak, ciąg znaków lub wyrażenie regularne. Wykonanie przykładowych instrukcji:

```
var tab1 = "a b c".split(" ");
var tab2 = "a.b.c".split(".", 2);
var tab3 = "a. b. c".split(".", 2);
var tab4 = "dwa".split("");
```

spowoduje utworzenie następujących tablic:

- ◆ tab1 — zawierającej trzy komórki z ciągami a, b i c — znakiem separatora jest bowiem spacja, a liczba ciągów wynikowych nie jest ograniczona,
- ◆ tab2 — zawierającej dwie komórki z ciągami a, b — znakiem separatora jest bowiem przecinek, a liczba ciągów wynikowych jest ograniczona do dwóch,
- ◆ tab3 — zawierającej dwie komórki z ciągami a i b — separatorem jest bowiem ciąg składający się z przecinka i spacji, a liczba ciągów wynikowych jest ograniczona do dwóch,
- ◆ tab4 — zawierającej trzy komórki z ciągami d, w i a — separatorem jest bowiem pusty ciąg znaków, co powoduje podzielenie ciągu na poszczególne litery (znaki).

Metoda substr, podobnie jak slice i substring, pozwala wyodrębnić fragment ciągu. Pierwszy argument określa indeks początkowy, a drugi — liczbę znaków do pobrania. Drugi argument można pominąć — wtedy pobierany fragment rozpocznie się od indeksu wskazywanego przez pierwszy argument, a skończy w końcu ciągu głównego. Znaczy to, że przykładowe wywołanie:

```
str = "wspaniały świat".substr(2, 4);
```

spowoduje przypisanie zmiennej str ciągu pani (4 znaki, począwszy od znaku o indeksie 2 w ciągu głównym), a wywołanie:

```
str = "wspaniały świat".substr(9);
```

spowoduje przypisanie zmiennej str ciągu świat (wszystkie znaki, począwszy od tego o indeksie 9, aż do końca ciągu głównego).

Metoda substring pobiera wybrany fragment ciągu wyznaczany przez indeksy wskazywane przez pierwszy i drugi argument. Jeżeli nie zostanie podany drugi argument, przyjmuje się, że pobrane zostaną wszystkie znaki, począwszy od indeksu wskazywanego

przez pierwszy argument, aż do końca ciągu. Pod tym względem zachowuje się tak samo jak `slice`. Jednak, w odróżnieniu od `slice`, jeżeli pierwszy argument jest większy od drugiego (czyli indeks początkowy jest większy od końcowego), ich kolejność zostanie zamieniona. Zatem wywołanie `substring(4, 2)` zostanie potraktowane jak `substring(2, 4)`. Przykładowe wywołania:

```
str1 = "wizja pięknego świata".substring(15);  
str2 = "wizja pięknego świata".substring(11, 14);  
str3 = "wizja pięknego świata".substring(20, 15);
```

spowodują:

- ◆ przypisanie zmiennej `str1` ciągu `świat`;
- ◆ przypisanie zmiennej `str2` ciągu `ego`;
- ◆ przypisanie zmiennej `str3` ciągu `świat`.

Działanie metod `toLowerCase` i `toUpperCase` jest analogiczne, choć działają przeciwnie. Pierwsza zamienia wszystkie litery ciągu na małe, a druga — na wielkie. Dzieje się to, niezależnie od tego, jaka była wielkość liter w ciągu oryginalnym. Zwracany jest ciąg przetworzony, a ciąg oryginalny nie jest zmieniany. Znaczy to, że instrukcja:

```
str1 = "Wielki Zderzacz Hadronów".toLowerCase();
```

spowodzi przypisanie zmiennej `str1` ciągu `wielki zderzacz hadronów`, a instrukcja

```
str2 = "Wielki Zderzacz Hadronów".toUpperCase();
```

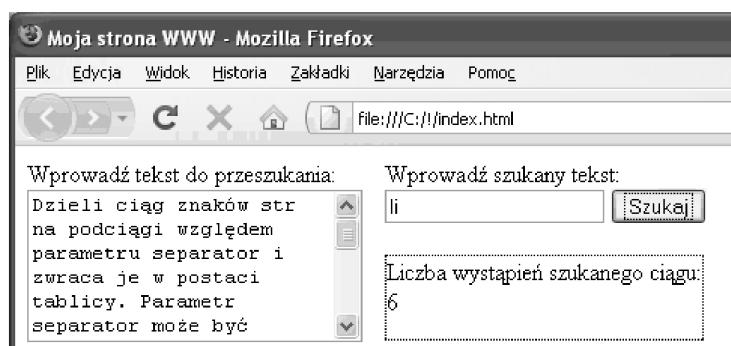
przypisanie zmiennej `str2` ciągu `WIELKI ZDERZACZ HADRONÓW`.

Użycie metod operujących na ciągach

Skoro poznaliśmy już cały zestaw metod operujących na ciągach znaków, użyjmy choć jednej z nich w praktycznie działającym przykładzie. Umożliwimy użytkownikowi witryny wprowadzenie dłuższego tekstu i automatyczne wliczenie, ile razy występuje w nim dowolna fraza — czyli dowolny ciąg znaków. Strona będzie wyglądała tak, jak zostało to zaprezentowane na rysunku 5.3. Znajduje się na niej rozszerzone pole tekstowe, zwykłe pole tekstowe, przycisk oraz warstwa prezentująca wyniki. Kod (X)HTML takiej witryny umieszczono na listingu 5.6.

Rysunek 5.3.

Witryna zliczająca liczbę wystąpień dowolnej frazy w danym tekście



Listing 5.6. Sekcja body kodu HTML opisanej witryny

```
<body>
    <div style="margin-right:15px;float:left;">
        <div>Wprowadź tekst do przeszukania:</div>
        <div><textarea id="ta1" cols="25" rows="5">
            </textarea></div>
    </div>
    <div style="float:left;">
        <div>Wprowadź szukany tekst:</div>
        <div style="margin-bottom:20px;">
            <input type="text" id="tf1">
            <input type="button" id="btn1"
                value="Szukaj" onclick="szukaj();">
        </div>
        <div id="divWynik"
            style="width:210px;height:55px;border:1px dotted black;">
        </div>
    </div>
</body>
```

Do formatowania układu strony zostały użyte warstwy — dwie główne tworzą układ kolumnowy. Na pierwszej znajduje się rozszerzone pole tekstowe zdefiniowane za pomocą znacznika `<textarea>`, któremu został nadany identyfikator `ta1`. Na drugiej umieszczone zostało zwykłe pole tekstowe o identyfikatorze `tf1`, przycisk oraz warstwa `divWynik` służąca do prezentacji wyników. Warstwa `divWynik` została wyróżniona przez zastosowanie kropkowanego (dotted) obramowania oraz ustalenie konkretnych rozmiarów (210×55 pikseli). Przyciskowi została przypisana procedura zdarzenia `onclick` w postaci funkcji `szukaj`, która zajmie się przeszukiwaniem tekstu wprowadzonego do pola `ta1`. Jej treść jest widoczna na listingu 5.7.

Listing 5.7. Treść funkcji `szukaj`

```
<script type="text/javascript">
    function szukaj()
    {
        var ta1 = document.getElementById('ta1');
        var tf1 = document.getElementById('tf1');
        var divWynik = document.getElementById('divWynik');
        var komunikat = "Liczba wystąpień szukanego ciągu: ";

        var tekst = ta1.value;
        var szukanyTekst = tf1.value;

        if((tekst == "") || (szukanyTekst == ""))
            return;

        var indeks = 0;
        var indeks_wystapienia = 0;
        var liczba_wystapieñ = 0;
```

```
while (indeks_wystąpienia != -1){  
    indeks_wystąpienia = tekst.indexOf(szukanyTekst, indeks);  
    if (indeks_wystąpienia != -1){  
        indeks = indeks_wystąpienia + 1;  
        liczba_wystąpień++;  
    }  
}  
divwynik.innerHTML = komunikat + liczba_wystąpień;  
}  
</script>
```

Funkcja pobiera odwołania do wszystkich niezbędnych elementów strony: rozszerzonego pola tekstuowego ta1, pola tekstuowego tf1 oraz warstwy divwynik i zapisuje je w zmiennych o takich samych nazwach. Definiuje również zmienną komunikat zawierającą podstawową treść wyświetlanego jako wynik komunikatu. W dalszej części skryptu zostanie ona uzupełniona o dane dotyczące liczby wystąpień poszukiwanego ciągu. Następnie funkcja odczytuje tekst do przeszukania (właściwość value pola ta1) oraz tekst poszukiwany (właściwość value pola tf1). Dane te zapisuje w zmiennych tekst oraz szukanyTekst.

W kolejnym kroku za pomocą instrukcji warunkowej if bada, czy któraś z tych zmiennych zawiera pusty ciąg znaków (if((tekst == "") || (szukanyTekst == "")). Jeśli tak jest, oznacza to, że albo nie ma czego przeszukiwać, albo nie ma czego poszukiwać. W takiej sytuacji działanie funkcji jest przerywane za pomocą instrukcji return. Jeżeli jednak istnieją teksty zarówno poszukiwany, jak i przeszukiwany, zapisowane są zmienne pomocnicze: indeks, indeks_wystąpienia i liczba_wystąpień. Ich znaczenie jest następujące:

- ◆ indeks — indeks znaku, od którego w danym kroku ma się zacząć przeszukiwanie tekstu głównego;
- ◆ indeks_wystąpienia — indeks, w którym w danym kroku został znaleziony poszukiwany ciąg znaków;
- ◆ liczba_wystąpień — całkowita liczba wystąpień poszukiwanego ciągu.

Przeszukiwanie tekstu odbywa się w pętli while. Jest wykonywana tak długo, jak dłużej zmienna indeks_wystąpienia jest różna od -1, czyli dopóty, dopóki w tekście przeszukiwanym można odnaleźć tekst poszukiwany. Pierwszą instrukcją wewnętrza pętli jest:

```
indeks_wystąpienia = tekst.indexOf(szukanyTekst, indeks);
```

To wywołanie metody indexOf dla ciągu zawartego w zmiennej tekst, czyli dla tekstu przeszukiwanego. Pierwszym argumentem jest wartość zmiennej szukanyTekst, czyli poszukiwany tekst, a drugim — wartość zmiennej indeks, czyli indeks znaku, od którego ma się rozpocząć przeszukiwanie (początkowo — 0).

Jeżeli indeks wystąpienia jest różny od -1:

```
if (indeks_wystąpienia != -1){
```

oznacza to, że tekst poszukiwany został odnaleziony na pozycji wskazywanej przez zmienną indeks_wystąpienia. Skoro tak, poszukiwanie kolejnego wystąpienia należy rozpocząć w następnym znaku, a więc trzeba ustawić zmienną indeks tak, by pokazywała kolejny znak za wskazywanym przez indeks_wystąpienia. (Czy aby na pewno jest to optymalne rozwiązanie? Można to sprawdzić w rozwiążaniu ćwiczenia 17.5). Ustawimy zatem zmienną indeks:

```
indeks = indeks_wystąpienia + 1;
```

oraz zwiększymy wartość zmiennej liczba_wystąpień o jeden:

```
liczba_wystąpień++;
```

Po zakończeniu pętli w zmiennej liczba_wystąpień będzie się znajdowała całkowita liczba wystąpień ciągu poszukiwanego w ciągu przeszukiwanym. Jest ona dodawana do komunikatu zapisanego w zmiennej komunikat, a całość jest przypisywana właściwości innerHTML warstwy divwynik. Dzięki temu wynik poszukiwań pojawia się na ekranie, co zostało zaprezentowane na znajdująącym się na początku tej części lekcji rysunku 5.3.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 17.1.

Zmień kod z listingu 5.3, tak by zamiast zdarzenia onkeyup było używane zdarzenie onkeydown. Sprawdź, jak zmieniło się działanie skryptu. Zastanów się, dlaczego wyniki nie są poprawne.

Ćwiczenie 17.2.

Zmień kod z listingu 5.5 tak, by nie korzystał z metod formatujących ciągi z tabeli 5.1, ale też nie zmienił się efekt działania skryptu.

Ćwiczenie 17.3.

W oparciu o przykład z listingów 5.4 i 5.5 napisz kod strony, która umożliwi zastosowanie do tekstu wprowadzonego do pola tekstowego dowolnej kombinacji czterech efektów generowanych przez metody formatujące. Zamiast pól wyboru typu radio użyj pól typu checkbox.

Ćwiczenie 17.4.

Napisz własną wersję metody substring. Nie używaj innych metod niż substr.

Ćwiczenie 17.5.

Kod z listingu 5.7 realizuje rodzaj algorytmu przeszukiwania liniowego. Zastanów się, czy jest optymalny, czy też można zwiększyć jego wydajność. Podpowiedź: przyjrzyj się dokładniej instrukcji `indeks = indeks_wystąpienia + 1;`, być może da się ją w prosty sposób usprawnić.

Lekcja 18. Wprowadzanie danych przez użytkownika

W lekcji 17. poznaliśmy rozmaite elementy witryny pozwalające na wybór różnych opcji bądź też wprowadzenie danych przez użytkowników. Wszystkie mogą być używane w formularzach WWW. Co prawda, dane z formularzy najczęściej przesyłane są do serwera, co już wykracza poza ramy tematyczne tej książki, jednak każdy programista JavaScript powinien wiedzieć, jak obsługiwać te elementy i sprawdzać poprawność uzyskiwanych za ich pomocą danych. W lekcji 18. omówimy więc formularze, sposoby walidacji danych, a także przetwarzanie informacji uzyskiwanych od użytkowników witryny.

Formularze

Formularz to typowy element witryny, który tworzymy, używając znacznika `form` z odpowiednimi atrybutami. Schematycznie wygląda to następująco:

```
<form name="nazwa"  
      target="okno"  
      action="url"  
      method="metoda"  
      enctype="typ_kodowania">  
  <!-- tutaj elementy formularza -->  
</form>
```

gdzie nazwa jest po prostu nazwą formularza. Parametr target określa nazwę okna, w którym ma się pojawić odpowiedź otrzymana z serwera, może on zawierać nazwę okna lub ramki, action ustala lokalizację serwera, do którego mają zostać wysłane dane zebrane z formularza, zwykle jest to adres skryptu PHP, ASP lub podobnego, method — sposób wysłania informacji do serwera (zwykle post lub get), natomiast enctype — sposób kodowania MIME.

Dostęp do obiektów formularzy zawartych na stronie HTML można uzyskać, odwołując się do właściwości `forms` obiektu `document` (tablica 4.4 w lekcji 12.). Całkowita liczbę formularzy uzyskamy, odczytując właściwość `length`, np.:

```
var liczba_formularzy = document.forms.length;
```

Natomiast dostęp do poszczególnych formularzy można uzyskać poprzez odwołania w postaci:

```
document.forms["nazwa_formularza"]
document.forms.nazwa_formularza
```

lub:

```
document.forms[indeks_formularza]
```

gdzie *nazwa_formularza* to nazwa zdefiniowana za pomocą atrybutu name, natomiast *indeks_formularza* to indeks kolejnego formularza w dokumencie (z reguły formularze indeksowane są w kolejności ich umieszczania w dokumencie HTML, indeks pierwszego formularza to 0). Jeżeli w definicji formularza użyjemy atrybutu id (nadamy mu identyfikator), odwołanie do obiektu formularza, oprócz wyżej wymienionych metod, będzie możliwa również uzyskać przy użyciu metody getElementById (tak samo jak w przypadku każdego innego elementu strony), np.:

```
var formularz = dokument.getElementById('identyfikator');
```

Obiekt formularza udostępnia właściwości zebrane w tabeli 5.4 i metody zebrane w tabeli 5.5.

Tabela 5.4. Właściwości obiektu formularza

Nazwa	Znaczenie
acceptCharset	Pobiera lub ustala listę typów kodowań znaków możliwych do zastosowania dla danych wprowadzanych do formularza (lista obsługiwanych stron kodowych).
action	Pobiera lub ustala wartość atrybutu action (określenie, gdzie mają zostać wysłane dane z formularza).
className	Pobiera lub ustala wartość atrybutu class.
dir	Pobiera lub ustala określenie kierunku tekstu.
encoding	Pierwotna nazwa właściwości enctype w starszych wersjach przeglądarek. Obecnie nie należy jej stosować.
enctype	Pobiera lub ustala typ MIME używany do kodowania zawartości formularza (wartość atrybutu enctype).
elements	Tablica zawierająca elementy składowe formularza.
id	Pobiera lub ustala identyfikator formularza (wartość atrybutu id).
lang	Pobiera lub ustala kod języka danego elementu.
length	Zawiera liczbę elementów składowych formularza.
method	Określa sposób wysyłania danych do serwera (wartość atrybutu method).
name	Pobiera lub ustala nazwę formularza (wartość atrybutu name).
target	Wartość atrybutu target, określenie okna lub ramki, do której mają zostać wysłane dane.
title	Pobiera lub ustala treść podpowiedzi dla formularza.

Tabela 5.5. Metody obiektu form

Metoda	Wywołanie	Opis
reset	reset()	Wykonuje reset formularza, czyli przywraca go do stanu wyjściowego. Działa więc tak samo jak przycisk <i>reset</i> .
submit	submit()	Powoduje wysłanie zawartości formularza do serwera. Działa tak samo jak przycisk <i>submit</i> .

Utwórzmy stronę zawierającą prosty formularz i napiszmy skrypt odczytujący podstawowe informacje o nim. Kod witryny przyjmie postać widoczną na listingu 5.8.

Listing 5.8. Strona zawierająca formularz

```
<body>
<form name="form_nr_1"
      action="http://moja.domena/form.php"
      method="get"
      style="width:250px;">
<div style="border: 1px solid black; margin:10px;
            width:250px; padding:10px; float:left;">
    <input type="radio" name="grp1" value="1" />Opcja nr 1
    <input type="radio" name="grp1" value="2" />Opcja nr 2
    <br>
    Wpisz tekst: <input type="text" name="txt1" value="" />
</div>
</form>
<div style="margin-bottom:10px;">
    <input type="button"
          value="Odczytaj dane dotyczące formularza"
          onclick="odczytaj();"/>
</div>
<div id="dataDiv" style="float:left;"></div>
</body>
```

Formularz zdefiniowano za pomocą znacznika `<form>`. Określona została jego nazwa (atribut `name` — `form_nr_1`), adres hipotetycznego skryptu PHP przetwarzającego dane (atribut `action` — `http://moja.domena/form.php`) oraz metoda wysyłania danych do serwera (atribut `method` — `get`). W formularzu znalazły się trzy elementy: dwa pola wyboru typu radio oraz zwykłe pole tekstowe. Wszystkie elementy zostały zdefiniowane za pomocą znacznika `<input>`.

Za formularzem znajduje się warstwa zawierająca przycisk. Kliknięcie przycisku będzie powodowało odczytanie informacji o formularzu oraz wyświetlenie ich na przeznaczonej do tego celu warstwie (ostatnia warstwa o identyfikatorze `dataDiv`). W związku z tym, zdarzeniu `onclick` przycisku została przypisana procedura obsługi w postaci funkcji `odczytaj`. Treść tej funkcji została przedstawiona na listingu 5.9.

Listing 5.9. Funkcja odczytująca informacje o formularzu

```
<script type="text/javascript">
    function odczytaj()
    {
        var form1 = document.forms["form_nr_1"];
        var tekst = "";
        tekst += "Nazwa formularza: ";
        tekst += form1.name + "<br />";
        tekst += "Liczba elementów formularza: ";
        tekst += form1.length + "<br />";
        tekst += "Adres skryptu przetwarzającego: ";
        tekst += form1.action + "<br />";
        tekst += "Typ kodowania: ";
        tekst += form1.enctype + "<br />";
        tekst += "Metoda wysyłania danych: ";
        tekst += form1.method + "<br />";
        var dataDiv = document.getElementById("dataDiv");
        dataDiv.innerHTML = tekst;
    }
</script>
```

Funkcja pobiera odwołanie do obiektu formularza. Robi to, odwołując się do właściwości forms obiektu document. Właściwość tę możemy potraktować jak tablicę, choć lepiej myśleć o niej jak o obiekcie zawierającym odwołania do wszystkich formularzy znajdujących się w dokumencie. Dostęp do konkretnego formularza uzyskujemy zatem, pisząc:

```
document.forms["nazwa_formularza"]
```

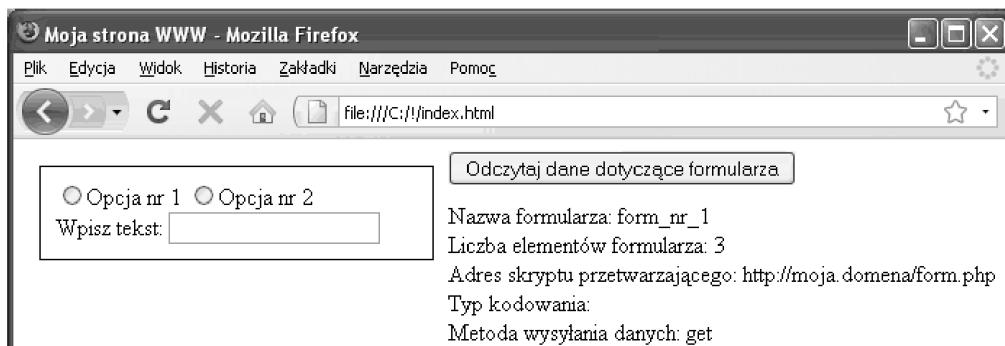
gdzie nazwa_formularza to nazwa nadana mu przez atrybut name. Ponieważ w naszym przypadku nazwa formularza to form_nr_1, odwołanie jest pobierane przy użyciu instrukcji:

```
var form1 = document.forms["form_nr_1"];
```

W ten sposób zmienna form1 zawiera obiekt formularza i już bez problemów możemy odczytać dotyczącego informacje. Pobieramy zatem jego nazwę (form1.name), liczbę elementów (form1.length), adres skryptu serwera (form1.action), typ kodowania danych (form1.enctype) oraz metodę wysyłania danych do serwera (form1.method). Odczytywane informacje są dodawane do zmiennej tekst, która ostatecznie jest przypisywana właściwości innerHTML warstwy dataDiv. To wszystko sprawia, że po kliknięciu przycisku na ekranie zobaczymy widok zaprezentowany na rysunku 5.4. Warte wspomnienia są jeszcze dwie rzeczy. Po pierwsze, liczbę elementów formularza można również odczytać w inny sposób. Skoro bowiem właściwość elements obiektu formularza to tablica zawierająca elementy składowe formularza, liczbę elementów można uzyskać za pomocą odwołania:

```
obiekt_formularza.elements.length
```

(w tym przypadku `form1.elements.length`). Po drugie, jak widać na rysunku 5.4., brakuje informacji o typie kodowania (właściwość `enctype` jest pusta — zawiera pusty ciąg znaków). Jest tak dlatego, że w definicji formularza nie został określony atrybut `enctype`².



Rysunek 5.4. Zostały odczytane informacje dotyczące formularza

Sprawdzanie poprawności danych

Jednym z najczęstszych zadań, związanych z formularzami, do których używany jest JavaScript, jest sprawdzanie poprawności danych wprowadzanych przez użytkownika. Czynność tę nazywamy walidacją formularzy. Co prawda, badanie poprawności trzeba również zawsze wykonać w skrypcie działającym po stronie serwera, jednak nie ma sensu wysyłanie danych do serwera, gdy są niekompletne lub błędne.

Jak badać dane wpisane do formularza? Sposoby są dwa. Pierwszy z nich wymaga, by nie umieszczać w formularzu elementu (przycisku) `submit`, ale zwyczajny przycisk generowany za pomocą znacznika `<input>` z atrybutem `type` ustawionym na `button`. Wysłaniem danych, po zbadaniu ich poprawności, będzie się musiała zająć procedura obsługi zdarzenia `onclick` tego przycisku. Użyjmy tego sposobu. Przygotujemy przykładowy prosty formularza rejestracyjny, zawierający pola pozwalające na wprowadzenie imienia, nazwiska, numeru telefonu i miasta. Dwa pierwsze z wymienionych pól będą wymagalne. Znaczy to, że użytkownik witryny będzie musiał wpisać do nich dane. Jeśli tego nie zrobi, przy próbie dokonania rejestracji zostanie wyświetlony komunikat o błędzie.

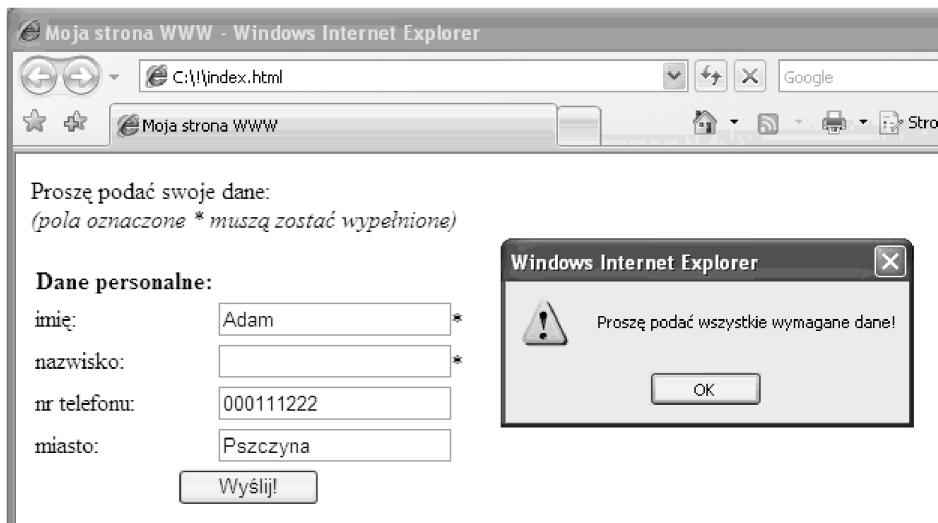
Kod (X)HTML takiej strony będzie miał postać przedstawioną na listingu 5.10.

² Jest tak w przeglądarkach Firefox i Opera. Internet Explorer uzupełnia brakującą wartość domyślnym typem kodowania: `application/x-www-form-urlencoded`.

Listing 5.10. Przykładowy formularz rejestracyjny

```
<body>
    <div style="margin-bottom:10px;">
        Proszę podać swoje dane:
        <br />
        <i>(pole oznaczone * muszą zostać wypełnione)</i>
    </div>
    <form name="form_nr_1" action="form.php" method="get">
        <table border="0">
            <tr>
                <td><b>Dane personalne:</b></td>
                <td></td>
            </tr><tr>
                <td>imię:</td>
                <td><input type="text" name="imię">*</td>
            </tr><tr>
                <td>nazwisko:</td>
                <td><input type="text" name="nazwisko">*</td>
            </tr><tr>
                <td>nr telefonu:</td>
                <td><input type="text" name="telefon"></td>
            </tr><tr>
                <td>miasto:</td>
                <td><input type="text" name="miasto"></td>
            </tr><tr>
                <td align="center" colspan="2">
                    <input type="button" name="wyslij"
                           value=" Wyślij! "
                           onclick="sprawdź_formularz();">
                </td>
            </tr>
        </table>
    </form>
</body>
```

Formularz został zdefiniowany za pomocą znacznika `<form>` z atrybutami `name`, `action` oraz `method`. Jego elementami są cztery pola tekstowe umożliwiające wprowadzenie imienia, nazwiska, numeru telefonu oraz miasta. Polom zostały nadane nazwy (wartości atrybutu `name`) zgodne z ich przeznaczeniem (`imię`, `nazwisko`, `telefon`, `miasto`). Dzięki temu będziemy mogli odwoływać się do nich z poziomu formularza. Formularz nie zawiera przycisku typu `submit`, ale zwykły przycisk generowany za pomocą znacznika `<input>` z atrybutem `type` ustawionym na `button`. Zdarzeniu `onclick` przycisku została przypisana procedura obsługi w postaci funkcji `sprawdź_formularz`. Jej zadaniem będzie sprawdzenie, czy wymagane pola `imię` i `nazwisko` zostały wypełnione. Jeśli tak, nastąpi wysłanie danych z formularza do serwera, jeśli nie, na ekranie pojawi się komunikat informujący o konieczności ich wypełnienia. Przykład zachowania witryny w przypadku niepodania nazwiska i kliknięcia przycisku `Wyślij` został zaprezentowany na rysunku 5.5, natomiast treść funkcji `sprawdź_formularz` — na listingu 5.11.



Rysunek 5.5. Niewypełnienie pola nazwisko spowodowało wyświetlenie komunikatu

Listing 5.11. Funkcja badająca poprawność danych

```
<script type="text/javascript">
    function sprawdź_formularz()
    {
        var form1 = document.forms["form_nr_1"];
        if(form1.imię.value == "" || 
            form1.nazwisko.value == ""){
            alert("Proszę podać wszystkie wymagane dane!");
            return;
        }
        form1.submit();
    }
</script>
```

Działanie funkcji jest bardzo proste. Pobiera odwołanie do formularza o nazwie `form_nr_1`, stosując znane już odwołanie w postaci:

```
var form1 = document.forms["form_nr_1"];
```

Po wykonaniu tej instrukcji zmienna `form1` zawiera obiekt formularza. Potrzebujemy również uzyskać dostęp do pól `imię` i `nazwisko`. Traktujemy te pola jak właściwości obiektu `form1`, używamy więc konstrukcji w postaci:

```
nazwa_formularza.nazwa_pola.value
```

Tym samym, wartość wpisaną w polu `imię` uzyskamy za pomocą odwołania:

```
form1.imię.value
```

a wartość zapisaną w polu `nazwisko` przy użyciu odwołania:

```
form1.nazwisko.value
```

Instrukcja warunkowa `if` pomoże zbadać, czy te wartości są pustymi ciągami znaków. Jeżeli którakolwiek z nich jest, za pomocą instrukcji alert wyświetlany jest komunikat informacyjny, a funkcja sprawdź_formularz kończy działanie przez wywołanie instrukcji `return`.

Jeżeli jednak wszystkie wymagane dane są obecne, trzeba wysłać dane do serwera, czyli w pewnym sensie zasymulować wcisnięcie przycisku `submit`. Robimy to, wywołując metodę `submit` (tabela 5.5) obiektu formularza:

```
form1.submit();
```

Istnieje też inny sposób dostępu do elementów formularza. Skoro możemy je traktować jak właściwości obiektu formularza — w tym przypadku obiektu `form1` — dostęp może być uzyskany z wykorzystaniem konstrukcji o postaci (lekcja 8.):

```
nazwa_formularza['nazwa_pola']
```

Zatem instrukcja warunkowa `if` z listingu 5.11 mogłaby też przyjąć postać:

```
if(form1['imie'].value == "" ||  
    form1['nazwisko'].value == ""){
```

Drugi sposób walidacji danych z formularza nie wymaga zamiany przycisku typu `submit` na zwykły, czyli formularz będzie miał w nim postać klasyczną. Jak wtedy wykryć, kiedy następuje wysłanie danych do serwera i jak temu przeciwodziąć w przypadku wykrycia niekompletności danych? Odpowiedź znajdziemy szybko, jeśli cofniemy się do lekcji 14. z rozdziału 4. i zajrzymy do tabeli 4.11 z listą zdarzeń. Odnajdziemy w niej zdarzenie `onsubmit`. Powstaje ono, gdy użytkownik witryny kliknie przycisk `submit` formularza. A więc można je wykorzystać do walidacji danych. Pozostaje problem poinformowania przeglądarki, czy formularz ma być wysłany, czy też nie. Wymaga to specyficznego przypisania procedury obsługi. Otóż, gdy będzie ona miała postać:

```
onsubmit="return false;"
```

formularz nie zostanie wysłany. Stąd już prosta droga do użycia funkcji walidującej dane:

```
onsubmit="return nazwa_funkcji();"
```

Jeśli teraz funkcja `nazwa_funkcji` zwróci wartość `false`, formularz nie zostanie wysłany, a jeżeli zwrócią jakkolwiek inną wartość (najlepiej `true`) — formularz zostanie wysłany. Zmodyfikujmy kod poprzedniego przykładu tak, by korzystał z nowo poznanej techniki. Kod (X)HTML przyjmie postać widoczną na listingu 5.12 (wspólna część kodu została zastąpiona komentarzami).

Listing 5.12. Formularz z elementem `submit`

```
<body>  
  <!-- tu wstaw początek kodu -->  
  <form name="form_nr_1" action="form.php" method="get"  
        onsubmit="return sprawdz_formularz();">  
    <table border="0">  
      <!-- tu wstaw definicje elementów -->
```

```
</tr><tr>
<td align="center" colspan="2">
    <input type="submit" name="wyslij"
           value=" Wyślij!" >
</td></tr>
</table>
</form>
</body>
```

Znacznik `<form>` otrzymał dodatkowy atrybut `onsubmit` w postaci:

```
onsubmit="return sprawdz_formularz();"
```

Znaczy to, że po kliknięciu przycisku `submit` zostanie wywołana funkcja `sprawdz_formularz`. Musi zwrócić wartość `false`, gdy dane nie są kompletne. Zmieniona została również zawartość ostatniej komórki tabeli formatującej elementy formularza. Zamiast przycisku generowanego przez znacznik `<input>` z atrybutem `type` ustawionym na `text` został użyty element typu `submit`, a więc również znacznik `<input>`, ale z atrybutem `type` ustawionym na `submit`.

Treść funkcji `sprawdz_formularz` została zaprezentowana na listingu 5.13.

Listing 5.13. Funkcja obsługująca zdarzenie `onsubmit`

```
<script type="text/javascript">
    function sprawdz_formularz()
    {
        var form1 = document.forms["form_nr_1"];
        if(form1.imie.value == "" || 
            form1.nazwisko.value == ""){
            alert("Proszę podać wszystkie wymagane dane!");
            return false;
        }
        return true;
    }
</script>
```

Ogólna postać funkcji jest bardzo podobna do tej z listingu 5.11. Tak samo pobierane jest odwołanie do formularza oraz jego elementów. Jednak po stwierdzeniu, że brakuje danych i wyświetleniu komunikatu, funkcja nie tylko kończy działanie, ale dodatkowo za pomocą instrukcji:

```
return false;
```

zwraca wartość `false`. Gdy z kolei wszystkie niezbędne dane są obecne, nie jest wykonywana metoda `submit`, ale jest zwracana wartość `true`. A zatem funkcja działa zgodnie z opisaną wyżej procedurą niezbędną do poprawnego obsłużenia zdarzenia `onsubmit`.

Wprowadzanie danych

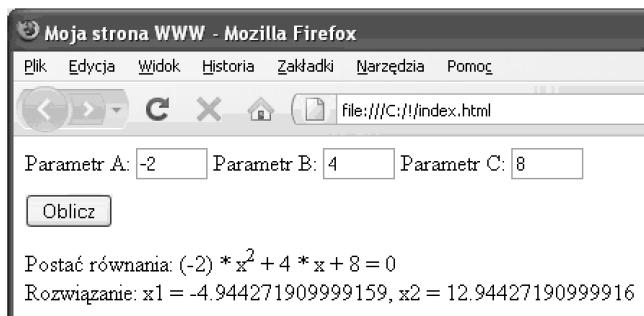
W lekcji 4., gdy poznawaliśmy instrukcje warunkowe, utworzyliśmy przykładowy skrypt rozwiązujący równania kwadratowe. Jego wadą była konieczność podawania parametrów równania bezpośrednio w kodzie programu — wtedy bowiem nie znaliśmy metod wprowadzania danych przez użytkowników. Teraz, gdy wiemy już, jak współpracować z przeglądarką i uzyskiwać dostęp do elementów witryny, z łatwością zbudujemy stronę, która pozwoli rozwiązywać takie równania o dowolnie wprowadzanych parametrach. Wystarczy przecież użyć kilku pól tekstowych i przycisku. Przykładowy kod (X)HTML takiej witryny został zaprezentowany na listingu 5.14, a efekt działania witryny na rysunku 5.6.

Listing 5.14. Witryna rozwiązuje równania kwadratowe

```
<body>
<div style="margin-bottom:10px;">
    Parametr A:
    <input type="text" id="tfA" size="4" />
    Parametr B:
    <input type="text" id="tfB" size="4" />
    Parametr C:
    <input type="text" id="tfC" size="4" />
</div>
<div style="margin-bottom:10px;">
    <input type="button" value="Oblicz"
          onclick="btnObliczClick();">
</div>
<div id="divWynik">
</div>
</body>
```

Rysunek 5.6.

Witryna pozwalająca na rozwiązywanie równań kwadratowych



Na witrynie zostały umieszczone trzy warstwy. Pierwsza zawiera trzy pola tekstowe, tfA, tfB i tfC, odpowiadające trzem parametrom równania: A , B i C . Równanie będzie bowiem przedstawiane w postaci klasycznej: $A * x^2 + B * x + C = 0$. Druga warstwa zawiera przycisk *Oblicz*. Została mu przypisana procedura obsługi zdarzenia *onclick* w postaci funkcji o nazwie *btnObliczClick*. Funkcja ta będzie wykonywana po kliknięciu przycisku. Trzecia warstwa ma identyfikator *divWynik* i początkowo jest pusta. Będzie używana do przedstawiania postaci równania oraz jego rozwiązań, co jest widoczne na rysunku 5.6.

Kod funkcji `btnObliczClick` został przedstawiony na listingu 5.15.

Listing 5.15. Treść funkcji `btnObliczClick`

```
function btnObliczClick()
{
    var A = document.getElementById("tfA").value;
    var B = document.getElementById("tfB").value;
    var C = document.getElementById("tfC").value;
    if(isNaN(A)){
        alert("Nieprawidłowy parametr A.");
        return;
    }
    if(isNaN(B)){
        alert("Nieprawidłowy parametr B.");
        return;
    }
    if(isNaN(C)){
        alert("Nieprawidłowy parametr C.");
        return;
    }
    var str = "Postać równania: ";
    str += A < 0 ? "(" + A + ")" : A;
    str += " * x2 + ";
    str += B < 0 ? "(" + B + ")" : B;
    str += " * x + ";
    str += C < 0 ? "(" + C + ")" : C;
    str += " = 0<br />";
    str += rozwiążRównanie(A, B, C);
    var divWynik = document.getElementById('divWynik');
    divWynik.innerHTML = str;
}
```

Zadaniem funkcji `btnObliczClick` jest odczytanie wprowadzonych do pól tekstowych danych, zbadanie, czy są to wartości liczbowe, sformowanie wzoru równania (do wyświetlenia na witrynie), wywołanie właściwej funkcji wykonującej obliczenia i wyświetlenie wyników na stronie. Odczytuje ona wartości wprowadzone do pól `tfA`, `tfB` i `tfC` i zapisuje je w zmiennych `A`, `B` i `C`. Następnie, korzystając z funkcji `isNaN` (lekcja 7.), bada, czy są to wartości numeryczne (mogą być interpretowane jako liczby). Jeżeli w którymkolwiek przypadku odpowiedź brzmi — nie, wyświetlana jest informacja o błędnych danych w tym polu tekstowym, a funkcja `btnObliczClick` kończy działanie przez wywołanie instrukcji `return`.

Jeżeli jednak w każdym z pól znajduje się wartość numeryczna, tworzona jest zmieniona `str`, której najpierw przypisywany jest ciąg opisujący równanie, a następnie wynik wywołania funkcji `rozwiążRównanie`, która wykonuje faktyczne obliczenia. W wywołaniu tej funkcji jako argumenty używane są zmienne `A`, `B` i `C` odzwierciedlające parametry równania. Na zakończenie zawartość zmiennej `str` jest przypisywana właściwości `innerHTML` warstwy `divWynik`, dzięki czemu zarówno równanie, jak i jego rozwiązanie pojawiają się na stronie.

Przyjrzyjmy się jeszcze, jak wygląda budowanie ciągu opisującego równanie. Nie wystarcza tu zwykłe dodanie do siebie ciągów składowych i wartości zmiennych. Gdybyśmy bowiem skonstruowali ten ciąg jako:

```
str += A + " * x2 + " + B + " * x + " + C + " = 0 ";
```

a parametry A , B i C były dodatnie, np. 2, 4, 6, w wyniku otrzymalibyśmy ciąg:

$$2 * x2 + 4 * x + 6 = 0$$

Określenie potęgi przy x powinno być jednak w indeksie górnym. Dlatego też do ciągu jest dodawany znacznik $\langle\sup\rangle$. To jednak też nie wystarcza, pozostaje jeszcze problem ujemnych parametrów. Gdyby konstrukcja ciągu miała postać:

```
str += A + " * x<sup>2</sup> + " + B + " * x + " + C + " = 0 ";
```

a parametry A , B i C wartości, np. -2, -4 i -6, wynik byłby następujący:

$$-2 * x^2 + -4 * x + -6 = 0$$

O ile pierwszy znak - możemy zaakceptować, o tyle kolejne są w tej postaci niedopuszczalne. Ten problem został rozwiązyany za pomocą nawiasów. Postać:

$$(-2) * x^2 + (-4) * x + (-6) = 0$$

jest poprawna. Wymagało to jednak badania stanu parametrów, które zostało wykonane za pomocą operatora warunkowego `:?` (lekcja 4.). A zatem jeżeli zostanie wykryte, że dany parametr (wartość danej zmiennej: A , B lub C) jest ujemny, jego wartość jest ujmowana w nawias okrągły. Oczywiście, kwestię tę można rozwiązać inaczej — rezygnując ze stawiania znaku + w przypadku ujemnych parametrów. Pozostawmy to jednak jako ćwiczenie do samodzielnego wykonania (ćwiczenie 18.2 na końcu lekcji).

Pozostała do napisania funkcja `rozwiążRównanie` dokonująca właściwych obliczeń. Jej treść została zaprezentowana na listingu 5.16.

Listing 5.16. Funkcja obliczająca równania kwadratowe

```
function rozwiążRównanie(A, B, C)
{
    if(isNaN(A) || isNaN(B) || isNaN(C)){
        return "Nieprawidłowe parametry.";
    }
    //sprawdzenie, czy jest to równanie kwadratowe
    if (A == 0){
        //A jest równe zero, równanie nie jest kwadratowe
        return "To nie jest równanie kwadratowe: A = 0!";
    }
    //A jest różne od zera, równanie jest kwadratowe
    //obliczenie delty
    delta = B * B - 4 * A * C;

    //jeśli delta mniejsza od zera
    if (delta < 0){
        var str = "To równanie nie ma rozwiązań w zbiorze ";
        str += "liczb rzeczywistych (delta < 0).";
        return str;
    }
}
```

```
//jeśli delta jest równa zero
if (delta == 0){
    //obliczenie wyniku
    var wynik = - B / 2 * A;
    return "Rozwiązanie: x = " + wynik;
}
//jeśli delta jest większa od zera
else{
    //obliczenie wyników
    wynik = (- B + Math.sqrt(delta)) / 2 * A;
    var str = "Rozwiązanie: x1 = " + wynik;
    wynik = (- B - Math.sqrt(delta)) / 2 * A;
    str += ", x2 = " + wynik;
    return str;
}
```

Funkcja przyjmuje trzy argumenty, A , B i C , odpowiadające parametrom równania, zwraca natomiast ciąg znaków opisujący wyniki. Na początku bada, czy argumenty są numeryczne, czyli zawierają wartości liczbowe. Choć może wydawać się niepotrzebne — takie badanie wykonywaliśmy przecież w funkcji `btn0blitzClick`, która wywołuje `rozwiążRównanie` — takie zachowanie jest niezbędne. W praktyce nie należy bowiem zakładać, że dane na pewno są poprawne (a co by się stało, gdybyśmy w przeszłości uzyli funkcji `rozwiążRównanie` w innym skrypcie, który nie dokonuje wstępnej weryfikacji danych?). Jeśli zatem funkcja wykryje, że któryś z parametrów jest błędny, zwraca ciąg z komunikatem o błędzie zamiast wyników.

Jeżeli jednak dane są prawidłowe, wykonywane jest obliczenie wyników. Czynność wykonywana jest na takiej samej zasadzie jak w lekcji 4. Oczywiście, sposób rozwiązywania równań kwadratowych się nie zmienił. Za pomocą serii instrukcji warunkowych badane są poszczególne możliwości ($\Delta < 0$, $\Delta = 0$, $\Delta > 0$) i w każdym z możliwych przypadków konstruowany jest ciąg zawierający odpowiedni komunikat, czy to z rozwiązaniem, czy też z informacją o braku rozwiązania.

Przetwarzanie stylów

W lekcji 16. w rozdziale 4. pojawił się przykład pozwalający na odczytywanie i przesywanie stylów CSS danego elementu strony. Jego pierwsza wersja pobierała i zapisywała wartość właściwości `style` warstwy (lub właściwości `cssText` obiektu `style` dla przeglądarki Internet Explorer). Wersja druga odczytywała poszczególne właściwości obiektu `style` i zapisywała je w osobnych oknach tekstowych. A gdyby tak połączyć te dwa przykłady? To znaczy odczytywać i przypisywać dane obiektu `style`, ale prezentować je tylko w jednym polu tekstowym. Wtedy skrypt działałby we wszystkich przeglądarkach i pozwalałby na jednoczesne zapisywanie wielu właściwości. To wymagałoby jednak przynajmniej prostej analizy ciągów wprowadzanych przez użytkownika witryny. Skoro jednak znamy już metody operujące na ciągach (lekcja 17.), możemy taki przykład wykonać. Kod (X)HTML oprzemy na przykładzie z listingu 4.44.

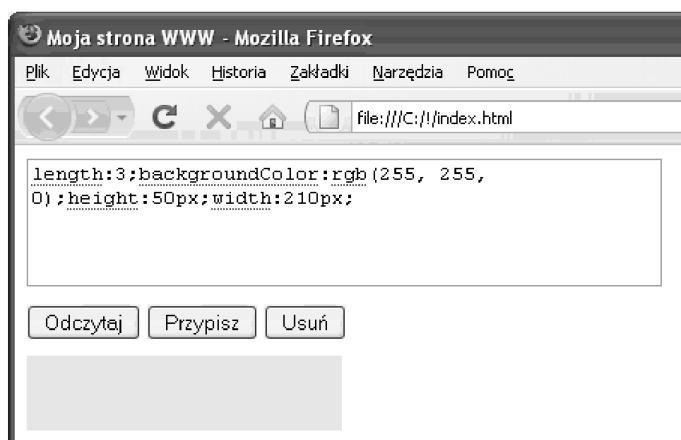
Zmienimy w nim jedynie pole tekstowe `ta1` na rozszerzone pole tekstowe, tak by łatwiej było objąć całość definicji stylu. W związku z tym, definicję warstwy zawierającej to pole zmieniamy na:

```
<div style="margin-bottom:10px;">
    <textarea id="ta1" cols="50" rows="4"></textarea>
</div>
```

Kod oryginalnego skryptu znajduje się na listingu 4.45. Trzeba w nim jednak całkowicie wymienić funkcje `odczytajStyl` i `przypiszStyl` (funkcja `usuńStyl` i odpowiadający jej przycisk nie są istotne dla bieżącego przykładu, ale można je pozostawić bez zmian).

Zadaniem funkcji `odczytajStyl` będzie pobranie nazw oraz wartości właściwości obiektu `style`, sformowanie z nich ciągu opisującego styl i wpisanie go do pola tekstopowego `ta1`. Po kliknięciu przycisku *Odczytaj* zobaczymy więc widok podobny do zaprezentowanego na rysunku 5.7. Odczytane właściwości mogą być różne, w zależności od zastosowanej przeglądarki, a nawet poszczególnych wersji tej samej przeglądarki. Ten skrypt umożliwia również proste badanie różnic w interpretacji stylów i zachowania obiektu `style` w różnych produktach. Internet Explorer wyświetli np. dane przedstawione na rysunku 5.8.

Rysunek 5.7.
Zawartość obiektu
`style` w przeglądarce
Firefox



Aby jednak dane ukazały się na ekranie, niezbędne jest napisanie kodu funkcji `odczytajStyl`. Jej treść została zaprezentowana na listingu 5.17.

Na początku funkcja pobiera odwołanie do rozszerzonego pola tekstopowego `ta1` oraz właściwości style warstwy `div3`. Czynności te wykonywane są standardowo. Definiowana jest również zmienna `tekst`, która będzie zawierała wszystkie odczytane dane sformatowane jako zestaw atrybutów CSS i ich wartości. Naszym zadaniem jest więc utworzenie ciągu o postaci:

```
atrybut1:wartość1;atrybut2:wartość2;...atrybutN:wartośćN;
```

Rysunek 5.8.

Zawartość obiektu
style w przeglądarce
Internet Explorer



Listing 5.17. Treść funkcji odczytującej zawartość obiektu style

```
function odczytajStyle()
{
    var ta1 = document.getElementById("ta1");
    var objStyle = document.getElementById("div3").style;
    var tekst = "";
    for(indeks in objStyle){
        if(typeof objStyle[indeks] != 'function'
           && isNaN(indeks) && objStyle[indeks]
           && indeks != "cssText"){
            tekst += indeks + ":" + objStyle[indeks] + ";";
        }
    }
    ta1.value = tekst;
}
```

Nie wydaje się to skomplikowane. Trzeba w pętli `for...in` odczytać nazwy właściwości obiektu style oraz ich wartości i sformować w pary:

`atrybut:wartość;`

To zadanie wykonałaby pętla o postaci:

```
for(indeks in objStyle){
    tekst += indeks + ":" + objStyle[indeks] + ";";
```

Nazwę każdej właściwości obiektu style (a więc danego atrybutu CSS) odczytamy ze zmiennej `indeks`, a jej wartość poznamy, odczytując wartość znajdującą się pod indeksem `indeks` w obiekcie `objStyle`:

`objStyle[indeks]`

Po co więc ta złożona instrukcja warunkowa z listingu 5.17? Otóż dlatego, że podstawa wersja pętli odczytałaby zdecydowanie zbyt dużo danych. Trzeba więc było je ograniczyć za pomocą instrukcji warunkowej `if`. Czemu danych byłoby zbyt wiele? Są cztery powody odzwierciedlane przez cztery warunki instrukcji `if`.

1. Właściwość `style` danego elementu strony jest obiektem. Znaczy to, że oprócz właściwości związań z atrybutami stylu CSS, może zawierać również metody. Nie chcemy ich jednak wyświetlać, chodzi nam wyłącznie o atrybuty. Dlatego pierwszy warunek to `typeof objStyle[indeks] != 'function'`. To on sprawia, że pominięte zostaną wszystkie metody obiektu `style`.
2. Obiekt `style` może być traktowany jak tablica atrybutów CSS. Dlatego też za pomocą pętli `for...in` każda właściwość zostanie odczytana dwukrotnie. Raz pod swoją własną nazwą, a drugi raz — jako indeks tablicy (czyli atrybut `width` raz będzie występował jako właściwość `width`, a raz jako indeks, np. 12). Indeksowanie numeryczne nie jest w tym przykładzie potrzebne i tylko zaburza właściwy układ atrybutów. Dlatego pomijamy wszystkie wartości zmiennej iteracyjnej `indeks`, które są numeryczne. Służy temu warunek `isNaN(indeks)`.
3. Część właściwości będzie istniała, ale nie będzie miała przypisanych wartości (będą zawierały puste ciągi znaków lub wartość `null`). Nie trzeba ich uwzględniać, zatem używamy warunku `objStyle[indeks]` (jeśli wyrażenie `objStyle[indeks]` będzie zawierało pusty ciąg znaków lub wartość `null`, zostanie przekonwertowane na wartość typu `boolean` — `false`).
4. Obiekt `style` w niektórych przeglądarkach (Firefox, InternetExplorer) zawiera niestandardową właściwość `cssText`. Znajdziemy w niej style przypisane elementowi strony w postaci, jaką sami chcemy uzyskać. Wyświetlenie wartości tej właściwości nie dość, że spowodowałoby zduplikowanie danych, to sprawiłoby poważne problemy przy operacji przypisania. Dlatego też pomijamy tę właściwość, stosując warunek `indeks != "cssText"`.

Skoro wiemy już, jak odczytać atrybuty stylu CSS z obiektu `style`, zajmijmy się funkcją `przypiszStyl`, która wykonuje czynność odwrotną, tzn. pobiera dane znajdujące się w polu tekstowym `ta1`, odczytuje nazwy i wartości poszczególnych atrybutów oraz przypisuje je obiektowi `style`. Treść tej funkcji została zaprezentowana na listingu 5.18.

Listing 5.18. Treść funkcji przypisującej atrybuty stylu CSS

```
function przypiszStyl()
{
    var stylTxt = document.getElementById("ta1").value;
    var div3Style = document.getElementById("div3").style;
    var tab = stylTxt.split(":");
    for(indeks in tab){
        var tab2 = tab[indeks].split(":");
        if(tab2.length == 2){
            try{
                div3Style[tab2[0]] = tab2[1];
            }
            catch(e){
            }
        }
    }
}
```

Zawartość pola tekstowego `ta1` jest pobierana i zapisywana w zmiennej `stylTxt`. Odwołanie do obiektu `style` warstwy `div3` jest natomiast przypisywane zmiennej `div3Style`. Wszystkie atrybuty i ich wartości były zapisane w polu tekstowym w postaci jednego ciągu i w takiej też postaci znajdują się w zmiennej `stylTxt`. Najpierw trzeba więc wydzielić wszystkie pary atrybut-wartość. Wykonamy to w prosty sposób za pomocą metody `split` (tabela 5.2 w lekcji 17.). Zauważmy, że wszystkie takie pary są odzielone znakiem średnika, należy go więc potraktować jako separator. Instrukcja:

```
var tab = stylTxt.split(":");
```

spowoduje zatem utworzenie tablicy `tab`, w której kolejnych komórkach znajdą się wszystkie pary:

```
atrybut:wartość
```

Skoro tak, tablicę tę można odczytać w pętli `for...in`. Skoro każda komórka tej tablicy zawiera parę atrybut-wartość, w każdym jej przebiegu trzeba ponownie użyć metody `split`, tym razem jako separatora używając znaku dwukropka:

```
var tab2 = tab[indeks].split(":");
```

W ten sposób otrzymujemy tablicę `tab2` o dwóch komórkach. Pierwsza (o indeksie 0) zawiera nazwę atrybutu, druga (o indeksie 1) zawiera wartość tego atrybutu. Za pomocą instrukcji `if` upewniamy się, że tablica `tab2` na pewno ma 2 komórki (inna ich liczba oznaczałaby błąd w danych) oraz dokonujemy przypisania:

```
div3Style[tab2[0]] = tab2[1];
```

W ten sposób właściwość obiektu `div3Style` o nazwie zawartej w komórce `tab2[0]` (nazwa atrybutu stylu CSS) otrzymuje wartość znajdująca się w komórce `tab2[1]` (wartość atrybutu stylu CSS). Instrukcja przypisania jest ujęta w blok `try...catch` na wypadek, gdyby w definicji stylu znalazła się wartość, której nie można przypisywać.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 18.1.

Zmodyfikuj kod z listingów 5.15 i 5.16 tak, by w przypadku podania argumentów niebędących wartościami numerycznymi błąd był sygnalizowany za pomocą techniki wyjątków.

Ćwiczenie 18.2.

Zmień kod z listingu 5.15, tak by w przypadku podania ujemnych parametrów równanie było wyświetlane bez nawiasów, ale w sposób matematycznie prawidłowy.

Ćwiczenie 18.3.

Napisz skrypt, który na podstawie wprowadzonych przez użytkownika danych będzie obliczał pole i obwód koła lub innej figury geometrycznej.

Lekcja 19. Wyrażenia regularne

Wyrażenia regularne (z ang. *regular expressions*) pozwalają na budowanie wzorców tekstowych (wzorców opisujących ciągi znaków). Dzięki nim oraz funkcjom udostępnianym przez JavaScript w łatwy sposób można badać, czy dany wzorzec znajduje się w jakimś ciągu, lub też wyszukiwać wystąpienia wzorca w ciągu. W JavaScriptie składnia wyrażeń regularnych jest oparta na języku Perl, a pełna standaryzacja nastąpiła w ECMAScript v3. W lekcji 19. omawiamy podstawy wyrażeń regularnych i sposoby ich stosowania w JavaScriptie.

Obiekt RegExp

Wyrażenia regularne w JavaScriptie są reprezentowane przez obiekty typu `RegExp`. Obiekt taki może powstać na dwa sposoby: przez formalne wywołanie konstruktora lub użycie literala określającego wyrażenie. W pierwszym przypadku stosowana jest konstrukcja o schematycznej postaci:

```
new RegExp(wzorzec, atrybuty)
```

Taki obiekt może być przypisany zmiennej:

```
var zmienna = new RegExp(wzorzec, atrybuty);
```

W przypadku drugim w kodzie należy umieścić literal (stałą napisową) o postaci:

```
/wzorzec/atrybuty
```

Taki literal również może być przypisany zmiennej:

```
var zmienna = /wzorzec/atrybuty
```

Obiekt `RegExp` posiada właściwości zebrane w tabeli 5.6 oraz metody zebrane w tabeli 5.7. Zanim jednak przyjrzymy się bliżej przedstawionym właściwościom i metodom, musimy poznać podstawowe zasady budowy wyrażeń regularnych.

Jak korzystać z wyrażeń?

Zobaczmy, jak korzystać w wyrażeń. Otóż, wyrażenie regularne tworzy wzorzec, który opisuje pewne ciągi znaków. Składa się ono ze zwykłych znaków, które są interpretowane bezpośrednio, oraz znaków i sekwencji znaków specjalnych, które mają specjalne znaczenie. Znakami specjalnymi zajmiemy się w kolejnej części lekcji, spróbujmy na razie utworzyć najprostsze wyrażenie. Może mieć ono postać:

```
/javascript/
```

Jest to wzorzec, który po prostu opisuje słowo `javascript`. Jeśli więc chcemy sprawdzić, czy pewien ciąg znaków zawiera to słowo, powinniśmy użyć metody `test`. Można to zrobić tak, jak zostało to przedstawione na listingu 5.19.

Tabela 5.6. Właściwości obiektu *RegExp*

Nazwa	Znaczenie	Dostępność
global	Zawiera wartość true, jeśli został ustawiony atrybut g, lub false w przeciwnym przypadku.	Od JavaScript 1.2 i JScript 3.0
ignoreCase	Zawiera wartość true, jeśli został ustawiony atrybut i, lub false w przeciwnym przypadku.	Od JavaScript 1.2 i JScript 3.0
input	Ciąg znaków, na którym przeprowadzana jest operacja dopasowania do wzorca.	Od JavaScript 1.2 i JScript 3.0
lastIndex	Pozycja ostatniego pasującego wzorca.	Od JavaScript 1.2 i JScript 3.0
lastMatch	Ostatni pasujący wzorzec.	Od JavaScript 1.2 i JScript 3.0
lastParen	Ostatnie pasujące do wzorca wyrażenie ujęte w nawiasy.	Od JavaScript 1.2 i JScript 3.0
leftContext	Tekst znajdujący się przed ostatnio odnalezionym wzorcem.	Od JavaScript 1.2 i JScript 3.0
multiline	Zawiera wartość true, jeśli został ustawiony atrybut m, lub false w przeciwnym przypadku.	Od JavaScript 1.2 i JScript 3.0
rightContext	Tekst znajdujący się za ostatnio odnalezionym wzorcem.	Od JavaScript 1.2 i JScript 3.0
source	Zawiera treść wyrażenia regularnego.	Od JavaScript 1.2 i JScript 3.0

Tabela 5.7. Metody obiektu *RegExp*

Metoda	Wywołanie	Znaczenie	Dostępność
compile →[. atrybuty])	<i>regexp.compile(wyrażenie</i>	Dokonuje wewnętrznej komplikacji wyrażenia <i>wyrażenie</i> , co pozwala na późniejsze szybsze jego przetwarzanie. Wyrażenie może zawierać atrybuty i, g, m, których znaczenie jest takie, jak w przypadku konstrukcji nowego wyrażenia.	Od JavaScript 1.2 i JScript 3.0
exec	<i>regexp.exec(str)</i>	Przeszukuje ciąg <i>str</i> w poszukiwaniu wzorca reprezentowanego przez obiekt <i>regexp</i> . Zwraca tablicę wynikową.	Od JavaScript 1.2 i JScript 3.0
test	<i>regexp.test(str)</i>	Sprawdza, czy w ciągu <i>str</i> jest zawarty wzorzec reprezentowany przez obiekt <i>regexp</i> . Jeśli tak, zwraca wartość true, jeśli nie, zwraca wartość false.	Od JavaScript 1.2 i JScript 3.0

Listing 5.19. Utworzenie wyrażenia regularnego za pomocą literalu

```
var wzorzec = /JavaScript/;
var str = "Czy znasz język JavaScript?";

if(wzorzec.test(str)){
    alert("Wskazany tekst zawiera wzorzec.");
}
else{
    alert("Wskazany tekst nie zawiera wzorca.");
}
```

Na początku zostały utworzone dwie zmienne: wzorzec i str. Pierwszej z nich została przypisany obiekt typu RegExp reprezentujący wzorzec JavaScript. Obiekt został utworzony za pomocą literalu `/JavaScript/`. Drugiej zmiennej został przypisany przykładowy ciąg znaków. Ponieważ zmienna wzorzec przechowuje obiekt typu RegExp, możliwe jest wywołanie jego metody o nazwie `test`. Jako argumentu tej metody należy użyć badanego ciągu znaków. Tym samym wywołanie:

```
wzorzec.test(str)
```

bada, czy w ciągu znaków znajdującym się w zmiennej str znajduje się wzorzec reprezentowany przez obiekt wzorzec. Jeśli tak jest, wywołanie metody zwróci wartość `true`, a w przeciwnym przypadku — wartość `false`. Wartość zwrócona przez metodę `test` jest badana za pomocą instrukcji warunkowej `if...else`. W tym przypadku ponieważ słowo `JavaScript` jest zawarte w ciągu `Czy znasz język JavaScript?`, wartością zwróconą przez `test` będzie `true`.

Badany ciąg znaków może, oczywiście, być przekazywany bezpośrednio jako argument metody `test` — korzystanie ze zmiennej pośredniczącej nie jest konieczne. Obiekt typu `RegExp` możemy też utworzyć przez użycie operatora `new` i formalne wywołanie konstruktora. Jak to zrobić, przedstawiono na listingu 5.20.

Listing 5.20. Utworzenie wyrażenia regularnego za pomocą konstruktora

```
var wzorzec = new RegExp("JavaScript");

if(wzorzec.test("Czy znasz język JavaScript?")){
    alert("Wskazany tekst zawiera wzorzec.");
}
else{
    alert("Wskazany tekst nie zawiera wzorca.");
}
```

Tym razem obiekt reprezentujący wyrażenie regularne jest tworzony za pomocą instrukcji:

```
new RegExp("JavaScript");
```

i przypisywany zmiennej `wzorzec`. Jak wiemy z lekcji 9. z rozdziału 3., jest to po prostu użycie operatora `new` i funkcji konstruującej (konstruktora) typu `RegExp`. Jako argument konstruktora przekazywany jest ciąg znaków zawierający wyrażenie regularne, z którego chcemy korzystać.

Dalsza część skryptu jest podobna do tego z listingu 5.19, z tą różnicą, że nie jest deklarowana pomocnicza zmienna przechowująca ciąg znaków, który chcemy przeszukiwać, ale ciąg jest przekazywany bezpośrednio w wywołaniu metody test:

```
wzorzec.test("Czy znasz język JavaScript?")
```

Warto również wiedzieć, że wywołanie metody obiektu RegExp może także dotyczyć składni korzystającej z literala. Można więc użyć przykładowej instrukcji o postaci:

```
var wynik = /JavaScript/.test("Czy znasz język JavaScript?");
```

Przypisuje ona zmiennej wynik rezultat działania metody test na obiekcie typu RegExp utworzonym za pomocą literala `/JavaScript/`.

W wyrażeniach regularnych można również stosować atrybuty (nazywane też *znacznikami* lub *flagami*), dodatkowo określające ich zachowanie. Mogą to być:

- ◆ `i` — oznaczający, że ma być ignorowana wielkość liter,
- ◆ `g` — oznaczający, że przetwarzanie ma dotyczyć całego ciągu,
- ◆ `m` — oznaczający, że przetwarzanie ma dotyczyć ciągu zawierającego wiele wierszy.

Atrybut `i` przydaje się, gdy wzorzec nie ma uwzględniać wielkości liter. Przykładowo wynikiem działania instrukcji:

```
/javascript/.test("Czy znasz język JavaScript?")
```

będzie wartość `false`, bowiem tekst Czy znasz język JavaScript? nie zawiera ciągu javascript (wielkość liter ma znaczenie). Jednak wykonanie instrukcji:

```
/javascript/i.test("Czy znasz język JavaScript?")
```

da w wyniku wartość `true`, bowiem dzięki atrybutowi `i` wielkość liter w przetwarzanym ciągu nie będzie miała znaczenia.

W analogiczny sposób użyjemy atrybutów podczas przypisywania wyrażeń zmiennym, np.:

```
var wzorzec = /JavaScript/i;
```

Jeżeli zaś chcemy użyć atrybutu przy tworzeniu wyrażenia za pomocą konstruktora, należy go przekazać jako drugi argument, np.:

```
var wzorzec = new RegExp("JavaScript", "i");
```

Można również stosować więcej niż jeden atrybut. Jeśli np. chcemy użyć zarówno `g`, jak i `i`, należy zastosować jedną z konstrukcji:

```
var wzorzec = /JavaScript/gi;  
var wzorzec = new RegExp("JavaScript", "gi");
```

Atrybut `g` nie ma znaczenia, jeśli jedynie badamy, czy tekst zawiera ciąg pasujący do wzorca, ma jednak znaczenie w przypadku przeszukiwania ciągu. Otóż, bez użycia `g` przeszukiwanie zakończy się po odnalezieniu pierwszego ciągu pasującego do wzorca (nawet jeśli w dalszej części tekstu znajdują się kolejne). Jeżeli jednak `g` zostanie użyty, tekst zostanie przeszukany w całości.

Budowanie wyrażeń

Znaki i metaznaki

Jak już wspominano, wyrażenia regularne budowane są ze znaków zwykłych oraz znaków specjalnych — metaznaków. Zwykłe znaki mają znaczenie dosłowne, czyli litera `a` we wzorcu odpowiada literze `a` w przetwarzanym ciągu. Metaznaki mają znaczenie dodatkowe. Omówimy je za chwilę. Do znaków zwykłych zaliczamy wszystkie litery alfabetu i cyfry, ale także znaki niealfabetyczne, które wymagają zastosowania sekwencji ucieczki (jak pamiętamy, technika ta była też używana w przypadku znaków specjalnych występujących w zwykłych ciągach znaków, tabela 2.1 w lekcji 2.). W wyrażenях można więc stosować sekwencje przedstawione w tabeli 5.8.

Tabela 5.8. Sekwencje ucieczki dla wyrażeń regularnych

Sekwencja	Znaczenie
\0	Bajt zerowy. Znak NUL o kodzie \u0000
\f	Nowa strona. Znak o kodzie \u000C
\n	Nowy wiersz. Znak o kodzie \u000A
\t	Tabulacja pozioma. Znak o kodzie \u0009
\r	Powrót karetki. Znak o kodzie \u000D
\v	Tabulacja pionowa. Znak o kodzie \u000B
\xNN	Znak ASCII w kodowaniu heksadecymalnym (szesnastkowym), np. \x92
\uNNNN	Znak Unicode w kodowaniu heksadecymalnym, np. \u0029
\cx	Sekwencja <code>Ctrl+znak</code> , np. ^C, ^J

Większość znaków interpunkcyjnych ma znaczenie specjalne i nie można używać ich bezpośrednio. Zaliczamy do nich: `^`, `$`, `.`, `*`, `+`, `?`, `=`, `!`, `:`, `|`, `\`, `/`, `(`, `)`, `[`, `]`, `{`, `}`. Jeżeli chcemy wykorzystać któryś z tych znaków, powinniśmy poprzedzić go znakiem `\` (lewy ukośnik). Przykładowo wyrażenie regularne zawierające znak dolara będzie miało postać:

`/\$/`

A zawierające znak zapytania wyrażenie JavaScript? należy zapisać jako:

`/JavaScript\?/`

Zasada ta dotyczy również samego znaku lewego ukośnika. Gdy chcemy napisać zawierające go wyrażenie, powinno ono mieć postać:

`/\\/`

Klasy znakowe

Ponieważ formowanie wyrażeń wyłącznie z pojedynczych znaków nie byłoby wygodne, mogą one tworzyć tzw. klasy znakowe. Klasa znakowa to po prostu pewien zestaw znaków. Klasę znakową możemy zbudować, korzystając z operatora zakresu, którym jest nawias kwadratowy. Istnieją także klasy predefiniowane. Zostały one zebrowane w tabeli 5.9.

Tabela 5.9. Klasy znakowe

Klasa	Znaczenie
[...]	Dowolny znak znajdujący się w nawiasie.
[^...]	Dowolny znak nieznajdujący się w nawiasie.
.	Dowolny znak, z wyjątkiem znaku nowego wiersza.
\d	Dowolna cyfra.
\D	Dowolny znak inny niż cyfra.
\s	Dowolny biały znak (spacja, tabulator itp.).
\S	Dowolny znak inny niż biały znak.
\w	Dowolny znak słowa (wyrazu) złożonego ze znaków ASCII.
\W	Dowolny znak niebędący znakiem słowa złożonego ze znaków ASCII.

Jak to działa w praktyce? Rozważmy znak kropki. Skoro reprezentuje ona każdy znak, z wyjątkiem znaku nowego wiersza, przykładowe wyrażenie:

/ko.ar/

będzie pasowało do ciągów: konar, kotar, kowar itp., czyli wszystkie poniższe przykładowe wywołania dadzą w wyniku wartość true (fragmenty pasujące do wzorca zostały dodatkowo wyróżnione):

```
/ko.ar/.test("Ten konar jest niebezpieczny.")
/ko.ar/.test("Ktoś stoi za kotarą.")
/ko.ar/.test("Obliczanie kowariancji nie jest trudne.")
/ko.ar/.test("Stoisz blisko artefaktu.")
```

Klasa \d pozwoli stwierdzić, czy w danym ciągu są jakieś cyfry. Przykładowo wywołanie:

/\d/.test("Konwaliowa 24");

da w wyniku wartość true, a:

/\d/.test("zwykły ciąg znaków");

da w wyniku wartość false.

Wyrażenie /\[ao]s/ będzie pasowało zarówno do ciągu los, jak i las (ale także ciągów losowy, laskowski itp.). Podobnie, jeśli chcemy przekonać się, czy w danym tekście występuje słowo serwer, ale nie wiemy, czy jest pisane przez w czy przez v, wystarczy użyć wzorca ser[wv]er.

Przydają się także zaprzeczenia. Gdy chcemy się upewnić, że w tekście są inne litery niż d, p i w, zastosujemy wzorzec:

/[^dpw]/

A gdy zapragniemy się dowiedzieć, czy w ciągu mamy inne znaki niż cyfry, przyda się wzorzec:

/[^\\d]/

Zakresy

Jeśli chcemy napisać wzorzec pasujący do każdej cyfry, to, zgodnie z opisem z poprzedniego podpunktu, możemy użyć wyrażenia:

/[0123456789]/

To nie byłoby wygodne. Jeszcze większy problem sprawiłoby nam wymienienie np. wszystkich liter alfabetu. Aby uniknąć takich sytuacji, możemy tego typu wyrażenie zapisać prościej. Wystarczy użyć znaku -. Pisząc zatem:

/[0-9]/

utworzymy wzorzec pasujący do każdej cyfry, a pisząc:

/[a-z]/

wzorzec pasujący do każdej małej litery.

Jeżeli interesują nas wszystkie litery, zarówno małe, jak i wielkie, napiszemy:

/[a-zA-Z]/

A gdy chodzi o wszystkie wielkie litery i cyfry:

/[A-Z0-9]/

lub:

/[A-Z\\d]/

Zakres nie musi być też „pełny”. Możemy np. wybrać cyfry od 3 do 8:

/[3-8]/

małe litery od f do k:

/[f-k]/

lub też cyfry od 2 do 5, małe litery od b do g i wielkie od K do O:

/[2-5b-gK-O]/

Powtórzenia

W wyrażeniach regularnych istnieją konstrukcje pozwalające na określenie liczby powtórzeń danego elementu. To duże ułatwienie. Założymy, że chcielibyśmy stwierdzić, czy w tekście znajduje się jakakolwiek liczba składająca się przynajmniej z czterech

cyfr (a dokładniej, czy istnieje podciąg składający się z przynajmniej czterech cyfr). Stosując dotychczas przedstawione techniki, trzeba by napisać wyrażenie:

`/[0-9][0-9][0-9][0-9]/`

lub:

`/\d\d\d\d/`

Lepiej nawet nie myśleć, jak w takiej sytuacji wyglądałyby bardziej złożone wzorce. Lepiej skorzystać z konstrukcji określających powtórzenia. Zostały one zebrane w tabeli 5.10.

Tabela 5.10. Wzorce powtórzeń

Wzorzec	Znaczenie
{n, x}	Co najmniej n powtórzeń, jednak nie więcej niż x.
{n,}	Co najmniej n powtórzeń.
{n}	Dokładnie n powtórzeń.
?	Zero lub jedno powtórzenie.
+	Co najmniej jedno powtórzenie.
*	Zero lub więcej powtórzeń.

Jak zatem powinien wyglądać wzorzec pasujący do dowolnych czterech cyfr? Najprościej można go napisać tak:

`/\d{4}/`

lub:

`/[0-9]{4}/`

Jeśli interesują nas grupy od trzech do pięciu cyfr, napiszemy:

`/\d{3,5}/`

Mogimy też sprawdzić, czy w danym tekście występuje ciąg `www`:

`/w{3}/`

bądź też ciąg JavaScript poprzedzony i zakończony co najmniej jednym białym znakiem:

`/\s+JavaScript\s+/`

Zachłanność wyrażeń

Sposób dopasowywania wzorca wyrażeń regularnych jest zachłanny (z ang. *greedy*). Oznacza to, że za każdym razem dopasowywana jest tak duża liczba znaków, jaka tylko jest możliwa (maksymalny możliwy wzorzec). Jeśli np. mamy tekst:

Moja strona `www`

i chcemy odszukać w nim ciąg odpowiadający wzorcowi:

`/w+/`

w wyniku otrzymamy ciąg WWW. Wspomniany wzorzec oznacza bowiem jedną lub więcej liter W. Jeśli ten wzorzec zastosujemy do ciągu:

WwwwWWWWWWWWWWWW

zostanie do niego dopasowany cały ten ciąg. Takie zachowanie nie zawsze jest pożądane. Rozważmy tekst:

Gant avel bet kaset betek an Douar Nevez

Załóżmy, że chcemy odnaleźć w nim ciąg znajdujący się między literami a i e. Wydaje się zatem, że powinniśmy zastosować wzorzec:

/a.+e/i

Symbol . oznacza dowolny znak, a symbol + — jedno bądź więcej jego powtórzeń. A więc jest to wzorzec, który dopasuje takie ciągi, jakie pomiędzy literami a i e mają jeden lub więcej dowolnych znaków (oprócz znaku końca wiersza). Atrybut i oznacza ignorowanie wielkości liter. Jaki zatem podciąg naszego tekstu pasuje do tego wyrażenia? Jeśli spodziewamy się, że jest to ant ave:

Gant avel bet kaset betek an Douar Nevez

to jesteśmy w błędzie. Zachłanny mechanizm porównywania wzorca dopasuje bowiem tak dużo, jak to tylko możliwe dowolnych znaków występujących za a, a przed e (pamiętaj, że e to także jest „dowolny” znak). A zatem wynikiem będzie ant avel bet kaset betek an Douar Neve:

Gant avel bet kaset betek an Douar Nevez

Wyrażenie trzeba więc jakoś poprawić. Można to zrobić za pomocą technik poznańnych dotychczas, to jednak pozostawmy jako ćwiczenie do samodzielnego wykonania (ćwiczenie 19.1 na końcu lekcji). Już w JavaScript 1.5 wyrażenie zachłanne można przemienić w niezachłanne (z ang. *nongreedy*, co tłumaczy się również jako leniwe). Jeśli wyłączymy zachłanność, dopasowywana będzie minimalna liczba znaków pasująca do wzorca. Osiągamy to przez zastosowanie znaku ?. Można go postawić po znakach: {}, ?, +, *, czyli stosować konstrukcje: {}?, ??, +?, *?. Jak to zadziała w naszych przykładach? Wzorzec:

/W+?/

zastosowany do ciągu:

Moja strona WWW

da w wyniku pojedynczą literę W (najmniejszy możliwy ciąg spełniający warunek wzorca). Podobnie ten wzorzec zastosowany do ciągu:

WwwwWWWWWWWWWWWW

da również pojedynczą literę W.

W drugim przykładzie z tego podpunktu lekcji wzorzec:

/a.+e?/i

zastosowany do ciągu:

Gant avel bet kaset betek an Douar Nevez

da w wyniku ciąg ant ave, czyli to, o co było naszym pierwotnym celem.

Musimy jednak pamiętać, że wzorzec jest zawsze dopasowywany od strony lewej do prawej, w wyniku zostaną więc zawarte wszystkie znaki pasujące z lewej strony. A zatem wzorzec:

`/ .+?w/i`

zastosowany do tekstu Piękna strona WWW, nie da w wyniku odstępu i znaku *W*, ale ciąg:

Piękna strona W

Podobnie wzorzec:

`/ .*?xy/`

zastosowany do ciągu xxxxxy, nie da w wyniku xy, ale xxxx. Zatem w obu tych przypadkach wyrażenia zachłanne i niezachłanne zostaną potraktowane tak samo.

Grupowanie

Grupowanie to traktowanie fragmentu ciągu jako całości, tak jakby był pojedynczym elementem. Używamy w tym celu znaków nawiasu okrągłego. Jeśli np. zastosujemy wzorzec:

`/ (xy)+/`

będzie pasował do każdego ciągu typu xy, xyxy, xyxyxy itd., czyli zawierającego jedno lub więcej powtórzeń następujących po sobie liter xy. Oto inny przykład. Jeśli utworzymy wzorzec:

`/ albo(wiem)? /`

będzie pasował do każdego ciągu zaczynającego się od albo, po którym występuje bądź nie człon wiem, czyli pasować będą ciągi: albo, albowiem. Jeżeli ten wzorzec zmodyfikujemy:

`/ albo(wiem)* /`

pasować będą ciągi: albo, albowiem, albowiemwiem itd.

Referencje (odwołania) w grupowaniu

Gdy korzystamy z grupowania, mamy do dyspozycji bardzo ciekawą właściwość — możliwość odwołania do ciągu odnalezionego w tekście. Otóż, każda grupa wyodrębniona za pomocą nawiasu okrągłego ma swój kolejny numer. W wyrażeniu można się do tej grupy odwołać za pomocą składni:

`\numer`

np.: \1, \5 itd. W sumie można uzyskać dziesięć takich odwołań (od \1 do \9). Wyobraźmy więc sobie, że chcemy w tekście wyszukać ciągi, które na początku mają dowolną kombinację zer i jedynek, po nich następuje dowolna liczba znaków innych niż cyfra, a które kończą się tą samą kombinacją zer i jedynek, jaką była na początku. Interesują nas ciągi typu:

```
01abc01
11halo11
0110xyz0110
```

Jak je wyodrębnić? W pierwszej chwili zapewne nasuwa się wyrażenie:

```
/([01]+[^d]+[01])+/
```

które, niestety, nie spełnia postawionego zadania. Pasują bowiem do niego wszystkie ciągi, które na początku i końcu mają dowolną kombinację zer i jedynek, a w środku co najmniej jeden znak niebędący cyfrą, np.: 100abc10, 00xy11 itp. Tymczasem na początku i końcu miała znajdować się taka sama kombinacja zer i jedynek. A zatem niezbędnie jest odwołanie do pierwszej odnalezionej kombinacji zer i jedynek. Możemy to zrobić w sposób następujący:

```
/(([01]+)[^d]+)\1/
```

Fragment wyrażenia [01]+ został ujęty w nawias okrągły, czyli zgrupowany. Na końcu zaś zostało umieszczone odwołanie do pierwszej występującej w wyrażeniu grupy, czyli w tym przypadku grupy zdefiniowanej jako [01]+. Tym samym zostaną odnalezione takie ciągi, które na początku mają dowolną grupę zer i jedynek (([01]+)), po nich następuje co najmniej jeden znak niebędący cyfrą ([^d]+) i kończą się taką samą grupą zer i jedynek, jaką była na początku (\1).

Jeżeli w wyrażeniu występuje więcej grup, będą one numerowane kolejno, np.:

```
/(#)(\d+)\2\1/
```

\1 odnosi się tu do pierwszej grupy, czyli dowolnej liczby znaków #, a \2 — do drugiej grupy, czyli dowolnej kombinacji cyfr. Tym samym do takiego wzorca pasują ciągi:

```
#11#
#101101#
##2323##
```

itd.

Jeżeli nie chcemy nadawać danej grupie numeru, należy użyć innej postaci grupowania (dostępnej w JavaScriptie od wersji 1.5), mianowicie:

```
(?:grupowane_znaki)
```

Rozważmy przykład wzorca:

```
/albo(?:wiem)?\s(([01]+)\s)\1/
```

Widać wyraźnie, że mamy tu dwie grupy. Pierwsza to (?:wiem), a druga — ([01]+). Pierwsza grupa, ze względu na zastosowaną składnię nie ma numeru, w związku z tym, druga ma numer 1 i można się do niej odwołać jako \1. Tym samym, to wyrażenie

będzie pasowało do ciągów, które zaczynają się od słowa albo, po którym występuje lub nie dowolna liczba słów wiem, po których następuje odstęp, za którym znajduje się dowolna liczba kombinacji zer i jedynek zakończona odstępem, po którym następuje powtórzenie wspomnianej kombinacji. Pasować będą zatem przykładowe ciągi:

albo 11 11
albowiem 010 010
albowiemwiem 1100 1100

itd.

Przy stosowaniu odwołań należy jeszcze zwrócić uwagę na grupy zagnieżdżone. W jednej grupie mogą się bowiem znajdować inne. Oto przykład zagnieżdżenia:

/((\d+)\s(\w+))/

Mamy tu do czynienia z jedną grupą zewnętrzną (to całe wyrażenie) oraz dwoma grupami wewnętrznymi: $(\d+)$ i $(\w+)$. W takiej sytuacji numeracja zaczyna się zawsze od grupy zewnętrznej, a grupy wewnętrzne otrzymują kolejne numery, począwszy od lewej. Tym samym numer 1 ma grupa $((\d+)\s(\w+))$, numer 2 — $(\d+)$, a numer 3 — $(\w+)$.

Alternatywy

Kolejną przydatną cechą wyrażeń jest możliwość stosowania alternatyw. Alternatywa jest określona przez znak $|$ i działa analogicznie do operatora sumy logicznej przedstawionego w lekcji 3. Po prostu jeśli napiszemy:

/a|b/

określimy w ten sposób ciąg składający się z litery a lub litery b . Jeżeli użyjemy wyrażenia:

/ab|cd/

będą do niego pasowały ciągi ab i cd, a zastosowanie wzorca:

/l(a|e|o)s/

pozwoli na wyodrębnienie wyrazów las, les, los.

Przy stosowaniu alternatyw należy pamiętać, że przetwarzane są one od strony lewej do prawej i jeżeli zostanie stwierdzona zgodność, pozostałe możliwości nie są już dopasowywane. To oznacza, że wzorzec:

/a|ab/

zastosowany do ciągu ab da w wyniku ciąg a. Podobnie, wzorzec:

/nie|niewiedza/

zastosowany do ciągu niewiedza, do w wyniku ciąg nie.

Kotwice

Dotychczas opisane składowe wyrażeń regularnych pozwalały na dopasowywanie wzorców występujących w dowolnym miejscu ciągu. Czasem chcielibyśmy jednak mieć możliwość wskazania konkretnie interesującego nas miejsca, np. początku lub końca tekstu. W takiej sytuacji możemy użyć tzw. kotwic (z ang. *anchors*). Znaki umożliwiające zakotwiczenie wzorca zostały zebrane w tabeli 5.11.

Tabela 5.11. Sekwencje zakotwiczeń

Sekwencja	Znaczenie
^	Początek ciągu (lub wiersza w wyrażeniu wielowierszowym).
\$	Koniec ciągu (lub wiersza w wyrażeniu wielowierszowym).
\b	Odstęp między słowami.
\B	Pozycja niebędąca odstępem między słowami.
(?=p)	Wymaganie, by kolejne znaki pasowały do wzorca p.
(?!p)	Wymaganie, by kolejne znaki nie pasowały do wzorca p.

Jeśli więc chcemy zbadać, czy na początku tekstu znajduje się cyfra, możemy użyć wzorca:

/^\d/

a gdy chcemy sprawdzić, czy na końcu linii znajduje się znak #:

#\\$/

Jeśli interesują nas wiersze, które zawierają wyłącznie znaki # (co najmniej jeden), pomoże wyrażenie:

/^#+\$/

a jeśli takie, które zawierają wyłącznie ciągi zer i jedynek:

/^[01]+\$/

Być może, zechcemy też odszukać słowa Java bądź java. Nie możemy wtedy zastosować wyrażenia:

/[Jj]ava/

bo pasuje ono również do takich ciągów jak JavaScript czy JavaScriptu. Pomoże tu kotwica \b, zastosujemy bowiem wzorzec:

/\b[Jj]ava\b/

Czasem interesują nas też słowa spełniające dane kryterium, ale znajdujące się przed jakimś specyficzny ciągiem. Przykładowo chcemy znaleźć wszystkie słowa Java lub java, po których występuje dowolna liczba odstępów, a za nimi słowo script lub Script. Użyjemy wtedy wyrażenia:

/[Jj]ava\s+(?=script)/

Gdybyśmy chcieli wyszukać wszystkie słowa Java lub java, po których bezpośrednio nie występują słowa Script lub script, skorzystamy z wzorca:

```
/[Jj]ava(?!Ss]cript)/
```

Metody używające wyrażeń regularnych

Jedną z metod, które pozwalają w praktyce korzystać z wyrażeń, już poznaliśmy. Była to metoda test obiektu typu RegExp, która pozwalała stwierdzić, czy w tekście znajduje się podciąg odpowiadający wzorcowi opisanemu przez wyrażenie regularne. Drugą z metod udostępnianych przez typ RegExp jest exec (tabela 5.7). Jej wywołanie ma postać:

```
regexp.exec(ciąg_znaków)
```

gdzie *regexp* to obiekt reprezentujący wyrażenie regularne, a *ciąg_znaków* to tekst, który chcemy badać w poszukiwaniu fragmentów pasujących do wzorca. Wynikiem działania metody exec jest albo tablica wynikowa (zawierająca rezultaty przeszukiwania), albo wartość null, jeśli w tekście nie został odnaleziony żaden ciąg pasujący do wzorca.

Najpierw omówmy działanie metody, gdy wyrażenie nie zawiera atrybutu g, czyli kiedy chcemy wyszukać jedynie pierwsze wystąpienie podciągu pasującego do wzorca. Wtedy pasujący ciąg, o ile zostanie odnaleziony, znajdzie się w komórce o indeksie 0. W komórce o indeksie 1 zostanie umieszczony podciąg odpowiadający pierwszemu grupowanemu wyrażeniu, w komórce o indeksie 2 — podciąg odpowiadający drugiemu grupowanemu wyrażeniu itd. Całkowita liczba komórek będzie zawarta, podobnie jak w każdej innej tablicy, we właściwości lenght. Oprócz tego, w tablicy znajdują się właściwość index zawierająca indeks znaku, w którym został znaleziony ciąg pasujący do wzorca, a także właściwość input zawierająca przeszukiwany ciąg znaków (argument metody exec).

Gdy metoda exec jest wywoływana na rzecz obiektu reprezentującego wyrażenie z ustalonym atrybutem g, jej zachowanie jest nieco inne. W takim przypadku musi umożliwić odszukanie wszystkich (a nie tylko pierwszego) występujących w tekście ciągów pasujących do wzorca. Postać tablicy wynikowej jest, co prawda, taka sama jak w wyrażeniach bez atrybutu g, ale przeszukiwanie ciągu rozpoczęyna się od znaku wskazywanego przez właściwość lastIndex. Jeżeli wzorzec zostanie odnaleziony, zwracana jest tablica, a właściwość lastIndex jest ustawiana na pierwszy znak za odnalezionym ciągiem. Jeżeli wzorzec nie zostanie odnaleziony, zwracana jest wartość null, a właściwość lastIndex — zerowana. Oznacza to, że przeszukiwanie globalne za pomocą metody exec można przeprowadzić w prosty sposób przy użyciu pętli while.

Aby w praktyce testować różne wyrażenie regularne i obserwować wyniki działania metody exec, napiszmy skrypt, który to umożliwi. Część (X)HTML została przedstawiona na listingu 5.21.

Listing 5.21. Strona pozwalająca na testowanie wyrażeń regularnych

```
<body>
<div style="float:left; margin-right:20px;">

    Wyrażenie regularne:<br />
    <input type="text" id="tfWyrazenie"
           style="width:200px;"/><br /><br />

    Atrybuty:
    <input type="checkbox" id="chbI"> i
    <input type="checkbox" id="chbG"> g
    <input type="checkbox" id="chbM"> m
    <br /><br />
    <input type="button" value="Sprawdź"
           onclick="sprawdz();">
</div>
<div>
    Przeszukiwany tekst:<br />
    <textarea cols="20" rows="5" id="taText"
              style="width:200px;"/></textarea>
    <br /><br />
</div>
<div id="dataDiv" style="width:420px; border:1px dotted black;">
</div>
</body>
```

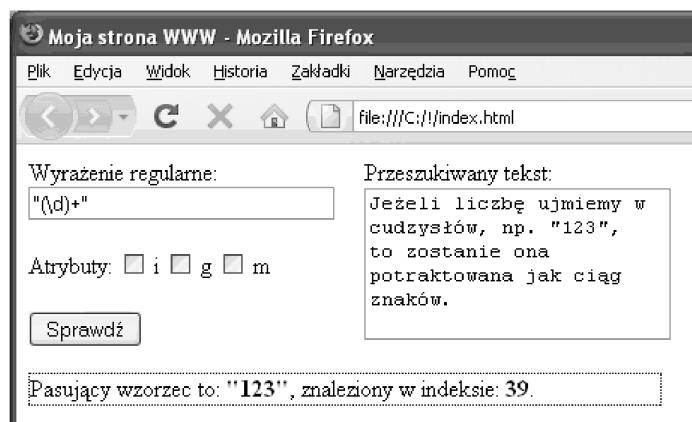
Kod składa się z trzech warstw. Pierwsza zawiera pole tekstowe pozwalające na wprowadzenie wyrażenia regularnego, trzy pola wyboru typu checkbox umożliwiające dodanie do wyrażenia atrybutów oraz przycisk wywołujący procedurę przeszukania tekstu w poszukiwaniu wzorca(ów) zdefiniowanego przez wyrażenie. Wszystkie elementy zostały zbudowane ze znaczników `<input>` z parametrem `type` ustawionym na odpowiedni typ (`text`, `checkbox` i `button`). Pole tekstowe oraz pola wyboru otrzymały identyfikatory (atrybut `id` znaczników `<input>`) umożliwiające odwołanie się do nich w kodzie skryptu (identyfikatory `tfWyrazenie`, `chbI`, `chbG` i `chbM`).

Druga warstwa zawiera rozszerzone pole tekstowe pozwalające na wprowadzenie tekstu, który ma być przeszukiwany. Pole to zostało zdefiniowane za pomocą znacznika `<textarea>` i został mu nadany identyfikator `taText`. Ponieważ pierwszej warstwie został nadany styl zawierający atrybut `float` o wartości `left`, druga warstwa znajdzie się po jej prawej stronie (rysunek 5.9).

Trzecia warstwa ma identyfikator `dataDiv` i początkowo jest pusta. To na niej będą się pojawiały wyniki działania skryptu. Wygląd witryny wraz z przykładem wyszukania wzorca jest widoczny na rysunku 5.9. Zastosowane zostało wyrażenie regularne `"(\d)+"`, które wyszuka takie fragmenty tekstu, jakie pomiędzy znakami cudzysłowu zawierają same cyfry (warto zastanowić się, czy trzeba było użyć grupowania, czy też wyrażenie to można z powodzeniem uproszczyć do `"\d+"`). Na trzeciej warstwie widoczny jest wynik wyszukiwania. Odnaleziony został ciąg "123" rozpoczynający się w indeksie 39. Czas więc napisać skrypt, który wykona postawione zadanie. Został przedstawiony na listingu 5.22. Jest to treść funkcji `sprawdz` wywoływanej po kliknięciu przycisku *Szukaj*.

Rysunek 5.9.

Efekt wyszukiwania wzorca we wprowadzonym tekście

**Listing 5.22.** Skrypt przeszukujący teksty po kątem występowania wskazanego wzorca

```
<script type="text/javascript">
function sprawdz()
{
    var wyrazenieTxt =
        document.getElementById("tfWyrazenie").value;
    var tekst =
        document.getElementById("taText").value;
    var dataDiv = document.getElementById("dataDiv");

    var attrib = "";
    if(document.getElementById("chbI").checked)
        attrib += "i";
    if(document.getElementById("chbG").checked)
        attrib += "g";
    if(document.getElementById("chbM").checked)
        attrib += "m";

    var wyrażenie = RegExp(wyrazenieTxt, attrib);
    var result;
    var str = "";

    if(!document.getElementById("chbG").checked){
        if((result = wyrażenie.exec(tekst)) != null){
            str += "Pasujący wzorzec to:<b> ";
            str += result[0];
            str += "</b>, znaleziony w indeksie: <b>";
            str += result.index;
            str += "</b>." ;
        }
    } else{
        str += "Wzorzec nie został odnaleziony.";
    }
    dataDiv.innerHTML = str;
}
```

```

else{
    var i = 1;
    while((result = wyrażenie.exec(tekst)) != null){
        str += "Pasujący wzorzec nr " + i++ + " to:<b> ";
        str += result[0];
        str += "</b>, znaleziony w indeksie: <b>";
        str += result.index;
        str += "</b>.";
        str += "Liczba odnalezionych grup: <b>";
        str += (result.length - 1) + "</b>: ";
        for(j = 1; j < result.length; j++){
            str += "grupa " + j + " = <b>";
            str += result[j] + "</b>:";
        }
        str += "<br />";
    }
    if(str == "") str = "Wzorzec nie został odnaleziony.";
    dataDiv.innerHTML = str;
}
</script>

```

Na początku pobierane są ciągi znaków wprowadzone w polach tfWyrażenie i taText oraz odwołanie do warstwy dataDiv. W tym celu jest używana standardowa metoda getElementById obiektu document. Dane są zapisywane w zmiennych wyrażenieTxt (wyrażenie regularne), tekst (tekst do przeszukania) i dataDiv (odwołanie do warstwy prezentacyjnej). Następnie konstruowana jest zmienna attrib, która będzie przechowywać atrybuty wyrażenia, oraz za pomocą instrukcji warunkowych if badany jest stan pól wyboru (wartość właściwości checked): chbI, chbG i chbM. Jeżeli którykolwiek z nich jest zaznaczone, do zmiennej attrib dodawana jest opowiadająca danemu polu litera (symbol atrybutu wyrażenia regularnego). A zatem, gdy zaznaczone jest pole chbI, do zmiennej attrib dodawana jest litera *i*, gdy zaznaczone jest pole chbG — litera *g*, a gdy zaznaczone jest pole chbM — litera *m*.

Po wykonaniu tych czynności wywoływany jest konstruktor obiektu typu RegExp, czyli budowany jest obiekt wyrażenia regularnego. Pierwszym argumentem konstruktora jest wartość odczytana z pola tfWyrażenie, a drugim — wartość zmiennej attrib. Nowo utworzony obiekt jest przypisywany zmiennej wyrażenie:

```
var wyrażenie = RegExp(wyrażenieTxt, attrib)
```

Po tej operacji deklarowane są dwie zmienne pomocnicze: result i str. Pierwsza z nich będzie przechowywać wynik zwrócony przez wywołanie metody exec, a druga — ostateczny komunikat, który ma się pojawić na stronie w rezultacie działania skryptu. Początkowym stanem zmiennej result jest undefined, a zmiennej str — pusty串 znaków.

Dalsze postępowanie jest zależne od tego, czy został zaznaczony atrybut *g*, czyli czy mają być wyszukiwane wszystkie ciągi pasujące do wzorca (atrybut *g* zaznaczony), czy też tylko pierwszy (atrybut *g* niezaznaczony). Jeżeli nie został zaznaczony:

```
if(!document.getElementById("chbG").checked){
```

postępowanie jest stosunkowo proste. Wywoływana jest bowiem metoda exec obiektu wyrażenie, a jako argument jest jej przekazywany ciąg znaków zapisany w zmiennej tekst. Wynik działania metody jest przypisywany zmiennej result:

```
result = wyrażenie.exec(tekst)
```

Aby nie wydłużać niepotrzebnie kodu, całe to wywołanie zostało ujęte w instrukcję warunkową if:

```
if((result = wyrażenie.exec(tekst)) != null){
```

dzięki czemu od razu badany jest jego wynik. Jeżeli jest to wartość null, wiemy, że w tekście nie występuje ciąg odpowiadający wzorcowi, zmiennej str jest więc przypisywany informujący o tym komunikat:

```
str += "Wzorzec nie został odnaleziony.;"
```

Jeżeli jednak wartość ta jest różna od null, ciąg został znaleziony i trzeba odczytać dane z tablicy result (jak pamiętamy, rezultatem działania metody exec jest tablica wynikowa). Odszukany ciąg pasujący do wzorca reprezentowanego przez obiekt wyrażenie znajdziemy pod indeksem 0:

```
str += result[0];
```

natomiast indeks, pod którym ta wartość się znajduje, we właściwości index:

```
str += result.index;
```

Dane wraz z pozostałym opisem są dopisywane do zmiennej str, która ostatecznie jest przypisywana właściwości innerHTML warstwy dataDiv:

```
dataDiv.innerHTML = str;
```

A to sprawia, że sformowany komunikat pojawi się na ekranie (co zostało zaprezentowane wyżej, na rysunku 5.9).

Gdy został zaznaczony atrybut g, czyli przeszukiwanie ma dotyczyć całego tekstu i mają zostać w nim odnalezione wszystkie ciągi pasujące do wzorca, postępowanie jest bardziej złożone. Ponieważ wyników może być dużo, są odczytywane w pętli while. Przed pętlą jest deklarowana zmienna i służąca jako licznik odszukanych ciągów:

```
var i = 1;
```

Warunek pętli jest taki sam jak w opisanej wyżej instrukcji if:

```
while((result = wyrażenie.exec(tekst)) != null){
```

Jest to więc wywołanie metody exec obiektu wyrażenie, przypisanie wyniku zmiennej result i porównanie go z wartością null. We wnętrzu pętli while następuje odczytanie uzyskanych danych. Pierwsza część jest podobna do poprzedniego przypadku, odszukany ciąg znajduje się bowiem w komórce o indeksie 0 (result[0]), a indeks jego wystąpienia we właściwości index. Różnica jest taka, że dodatkowo prezentowany jest numer odszukanego ciągu, który jest zapisany w zmiennej i:

```
str += "Pasujący wzorzec nr " + i++ + " to:<b> ";
```

Dodatkowo są jednak odczytywane dane dotyczące grupowania, które pominęliśmy przy przeszukiwaniu nieglobalnym. Otóż, możliwe jest odczytanie ciągów odpowiadających poszczególnym grupom zawartym w wyrażeniu regularnym. Są one zawarte w indeksach od 1 wzwyż, o ile — oczywiście — istnieje co najmniej jedna grupa. Najpierw odczytujemy więc całkowitą liczbę grup, która wynika ze wzoru *długość_tablicy - 1*:

```
str += (result.length - 1) + "</b>: ";
```

Następnie w pętli for odczytujemy ciągi odpowiadające każdej grupie:

```
for(j = 1; j < result.length; j++){
    str += "grupa " + j + " = <b>";
    str += result[j] + "</b>:";
```

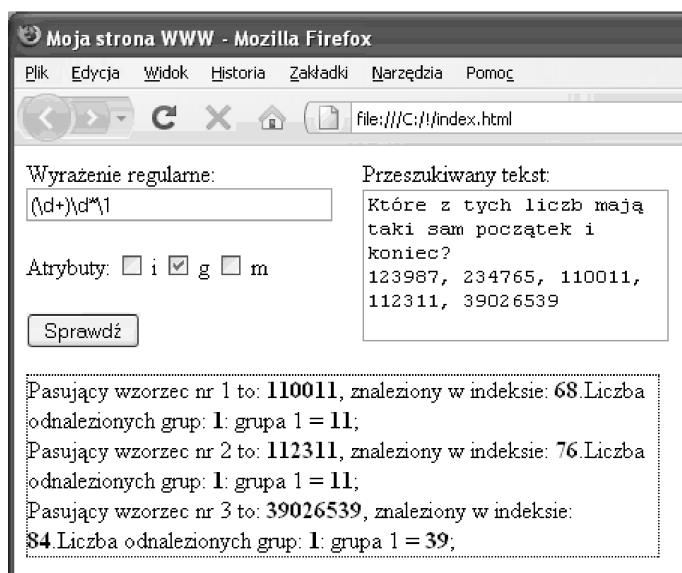
```
}
```

Po zakończeniu pętli while badamy, czy została wykonana co najmniej raz. Będzie tak, gdy zmienna str nie zawiera pustego ciągu znaków. Jeśli więc nie zawiera pustego ciągu (zawiera jakieś dane), jest przypisywana właściwości innerHTML obiektu dataDiv i informacje pojawiają się na ekranie (rysunek 5.10). Jeśli zawiera pusty ciąg znaków, oznacza to, że już pierwsze wywołanie metody exec zwróciło wartość null, czyli w tekście nie ma żadnego ciągu pasującego do wzorca. W takim przypadku zmiennej str przypisywany jest informujący o tym komunikat:

```
if(str == "") str = "Wzorzec nie został odnaleziony.:";
```

Jak w praktyce zadziała wyszukiwanie z włączoną opcją g, możemy zobaczyć na rysunku 5.10.

Rysunek 5.10.
Wynik globalnego
przeszukiwania tekstu



Zastosowane wyrażenie ma postać `(\d+)\d*\1`, a zatem chcemy odszukać w tekście takie ciągi cyfr, które zaczynają się i kończą takim samym zestawem znaków. Zapewnia to zgrupowanie pierwszego członu wyrażenia `((\d+))` oraz odwołanie do pierwszej grupy na końcu `(\1)`. Oznacza to również, że w każdym pasującym ciągu będzie jedna grupa. We wprowadzonym tekście do takiego wzorca pasują ciągi 110011 (grupa 11), 112311 (grupa 11) i 39026539 (grupa 39) i takie też wyniki pojawiły się na ekranie.

Na zakończenie tej części lekcji zwróćmy uwagę, że w zaprezentowanym przykładzie nie badamy, czy wyrażenie regularne jest poprawne. Jeśli nie będzie, skrypt nie działa, a na konsoli błędów zostanie zgłoszony błąd. Warto poprawić to niedopatrzenie, pozostawmy to jednak jako ćwiczenie do samodzielnego wykonania (ćwiczenie 19.2 na końcu lekcji).

Wyrażenia regularne i typ łańcuchowy

Wyrażenia regularne mogą być również stosowane w niektórych metodach obiektów typu String. Są to search, replace, match i split. Wszystkie zostały wymienione w tabeli 5.2 i pokróćce omówione w lekcji 17. Metoda search przyjmuje jako argument wyrażenie regularne (obiekt RegExp) i zwraca indeks wystąpienia ciągu odpowiadającego wzorcowi lub wartość -1, jeśli żaden fragment tekstu nie pasuje do wzorca. Jeżeli przekazany zostanie argument niebędący wyrażeniem regularnym, np. ciąg znaków, zostanie przekonwertowany do wyrażenia regularnego. Uwaga: metoda search ignoruje atrybut g wyrażenia, co oznacza, że nigdy nie przeprowadza przeszukiwania globalnego i odnajduje jedynie pierwsze wystąpienie ciągu odpowiadającego wzorcowi.

Metoda replace przyjmuje dwa argumenty: pierwszy określa poszukiwany ciąg, który zostanie zamieniony na ciąg przekazany w postaci drugiego argumentu. Metoda zwraca przetworzony tekst, ciąg oryginalny nie jest zmieniany. Pierwszy argument może być wyrażeniem regularnym. Jeżeli zostanie w nim zastosowany atrybut g, zamienione zostaną wszystkie ciągi pasujące do wzorca. Jeśli atrybut g zostanie pominięty, zostanie zamieniony tylko pierwszy ciąg. Przykładowe wywołanie:

```
"Kot jest szary. Kot ma ogon.".replace(/Kot/, "Pies");
```

spowoduje zwrócenie ciągu o postaci:

```
Pies jest szary. Kot ma ogon.
```

a wywołanie:

```
"Kot jest szary. Kot ma ogon.".replace(/Kot/g, "Pies");
```

spowoduje zwrócenie tekstu:

```
Pies jest szary. Pies ma ogon.
```

Jeżeli pierwszy argument metody replace nie jest wyrażeniem regularnym, tylko ciągiem znaków, wykonywana jest bezpośrednia zamiana ciągów (nie jest podejmowana konwersja pierwszego argumentu na wyrażenie regularne). Drugi argument może zawierać odwołania do grup występujących w wyrażeniu regularnym. Numer grupy należy poprzedzić znakiem \$. Jeśli np. chcemy wszystkie występujące w tekście ciągi cyfr ująć w znaki cudzysłowu, możemy użyć następującego wywołania:

```
replace(/(\d+)/g, "\"$1\"")
```

Wywołanie zastosowane do tekstu:

Takie oto liczby: 23, 22, 18

zmieni go w sposób następujący:

Takie oto liczby: "23", "22", "18"

Metoda `match` jako argument przyjmuje wyrażenie regularne i wyszukuje w tekście ciągi opisane przez to wyrażenie. Jej zachowanie jest różne, w zależności od tego, czy w wyrażeniu został zastosowany atrybut `g`, czy też nie. Jeżeli używane jest dopasowywanie globalne (atrribut `g` obecny), metoda zwraca tablicę zawierającą wszystkie ciągi pasujące do wzorca. Przykładowo wywołanie:

```
var tab = "6a5b4c".match(/(\d+)/g);
```

da w wyniku tablicę, której pierwsza komórka (o indeksie 0) zawiera wartość 6, druga — 5, a trzecia — 4. Liczbę komórek wynikowych (czyli liczbę odnalezionych ciągów) można odczytać, odwołując się do właściwości `length`. Oznacza to, że odczyt danych jest możliwy np. za pomocą pętli `for`:

```
var tab = "6a5b4c".match(/(\d+)/g);

var ile = tab.length;
for(i = 0; i < ile; i++){
    str += "Ciąg nr " + i + " = ";
    str += tab[i] + "<br />"
}
```

Jeżeli przeprowadzane jest przeszukiwanie nieglobalne (brak atrybutu `g`), zachowanie jest inne. Zwracana jest również tablica, jednak w pierwszej komórce zawiera ona odszukany ciąg pasujący do wzorca, a w kolejnych komórkach — ciągi odpowiadające kolejnym grupom wyrażenia. Dodatkowo zawiera ona właściwości `index` i `input`. Ma więc taką postać jak omówiona wyżej metoda `exec` obiektu `RegExp`.

Niezależnie od atrybutów wyrażenia regularnego, jeżeli w przeszukiwanym tekście nie występuje żaden ciąg pasujący do wzorca, zwracana jest wartość `null`.

Przykład, w którym pokazano sposób odczytania tablicy zwróconej przez metodę `match` operującą na wyrażeniu nieglobalnym, został zaprezentowany na listingu 5.23.

Listing 5.23. Odczyt tablicy zwróconej przez metodę `match`

```
var str = "";
var tab = "xyz61af52be43cf".match(/(\d+)([a-z]+)/i);

if(tab != null){
    str += "Odszukany został ciąg: ";
    str += tab[0];
    str += " występujący w indeksie: ";
    str += tab.index + "<br />";

    var ile = tab.length;
    for(i = 1; i < ile; i++){
        str += "Grupa nr " + i + " = ";
        str += tab[i] + "<br />";
    }
}
```

```
    }
} else{
    str += "Poszukiwany wzorzec nie został odnaleziony.";
}

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Ostatnią metodą obiektów typu String, która pozwala na użycie wyrażenia regularnego, jest `split`. Jak wiemy z lekcji 17., dzieli ona ciąg ze względu na znaki separatora. Jeśli jako separatora użyjemy wyrażenia regularnego, zyskamy wiele większe możliwości niż gdyby to był zwykły ciąg znaków. Jeśli separatorem ma być np. przecinek lub kreska pionowa albo dwukropki, możemy użyć wywołania:

```
var tab = "ab|cd:ef.gh".split(/[:.|\]/);
```

A gdy separatorem ma być dowolna liczba znaków odstępu skorzystamy z wywołania:

```
var tab = "ab cd ef gh".split(/\s+/);
```

Wrażenia regularne w praktyce

W lekcji 18. prezentowany był przykład pokazujący, jak można wstępnie weryfikować dane wprowadzone do formularza umieszczonego na stronie WWW (listingi 5.10 i 5.11). Wymagaliśmy wtedy, aby wypełnione zostały przynajmniej pola *imię* i *nazwisko*, a weryfikacja polegała na zbadaniu, czy w którymś z pól znajduje się pusty ciąg znaków. Gdy tak było, na ekranie pojawiał się stosowny komunikat. Trzeba jednak przyznać, że była to wyjątkowo prosta technika. Teraz, kiedy znamy już wyrażenia regularne i metody umożliwiające ich wykorzystywanie, możemy pokusić się o dużo bardziej wyrafinowaną weryfikację danych. Nic już nie stoi na przeszkodzie, abyśmy wymagali, by informacje były wprowadzane w ścisłe określonym formacie. W oparciu o przykład z lekcji 18. napiszmy kolejny. Tym razem formularz będzie zawierał pola dla imienia, nazwiska, telefonu oraz kodu pocztowego i wszystkie będą musiały być wypełnione. Format danych każdego z pól będzie weryfikowany przez odpowiednią procedurę JavaScript. Kod (X)HTML zawierający taki formularz został przedstawiony na listingu 5.24.

Listing 5.24. Formularz wymagający wprowadzania danych w określonym formacie

```
<body>
<div style="margin-bottom:10px;">
    Proszę podać swoje dane:
    <br />
    <i>(wszystkie pola muszą zostać wypełnione)</i>
</div>
<form name="form_nr_1" action="form.php" method="get">
<table border="0">
<tr>
    <td><b>Dane personalne:</b></td>
    <td></td>
```

```

</tr><tr>
    <td>imię:</td>
    <td><input type="text" name="imię"></td>
</tr><tr>
    <td>nazwisko:</td>
    <td><input type="text" name="nazwisko"></td>
</tr><tr>
    <td>nr telefonu:</td>
    <td><input type="text" name="telefon"> format +48 (022) 0000000</td>
</tr><tr>
    <td>kod pocztowy:</td>
    <td><input type="text" name="kod"> format 00-000</td>
</tr><tr>
    <td align="center" colspan="2">
        <input type="button" name="wyslij"
               value=" Wyślij! "
               onclick="sprawdź_formularz();">
    </td></tr>
</table>
</form>
</body>

```

Ogólna struktura kodu jest taka jak w przykładzie z listingu 5.10. Pole *miasto* zostało zmienione na *kod*. Zmianie uległ również opis formularza, dodane zostały informacje o formacie wprowadzanych danych dla pól *telefon* i *kod*. Po wczytaniu do przeglądarki formularz będzie miał postać przedstawioną na rysunku 5.11. Jak widać, numer telefonu będzie musiał być wprowadzony w formacie:

+kod_kraju (nr_kierunkowy) nr_właściwy

przy czym kod kraju będzie musiał mieć 2 cyfry, numer kierunkowy — również 2, a numer właściwy — 7. Kod pocztowy przyjmie zaś format:

dwie_cyfry-trzy_cyfry

Dodatkowo będziemy wymagać, aby w polach *imię* i *nazwisko* wprowadzone mogły być tylko litery, a każde z nich zawierało co najmniej dwa znaki. W związku z tym, dokonująca weryfikacji danych funkcja *sprawdź_formularz* przyjmie postać przedstawioną na listingu 5.25.

Na początku tworzone są wyrażenia regularne opisujące wzorce dla danych każdego z pól. Dla telefonu jest to:

```
var telefonRE = /\+\d{2} \(\d{2}\) \d{7}/;
```

Dla kodu pocztowego:

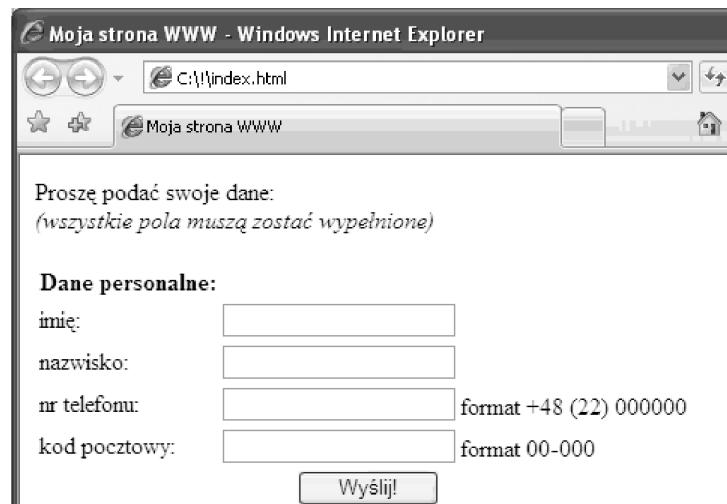
```
var kodRE = /\d{2}-\d{3}/;
```

Dla imienia i nazwiska:

```
var imięNazwRE = /[a-zA-Z]{2,}/;
```

Rysunek 5.11.

Formularz służący do podania danych personalnych

**Listing 5.25.** Funkcja weryfikująca dane, korzystająca z wyrażeń regularnych

```
<script type="text/javascript">
    function sprawdź_formularz()
    {
        var telefonRE = /\+\d{2} \(\d{2}\) \d{7}/;
        var kodRE = /\d{2}-\d{3}/;
        var imięNazwRE = /[a-zA-Z]{2,}/;

        var form1 = document.forms["form_nr_1"];

        var imięTxt = form1.imię.value;
        var nazwiskoTxt = form1.nazwisko.value;
        var telefonTxt = form1.telefon.value;
        var kodTxt = form1.kod.value;

        if(telefonRE.exec(telefonTxt) != telefonTxt ||
            kodRE.exec(kodTxt) != kodTxt ||
            imięNazwRE.exec(imięTxt) != imięTxt ||
            imięNazwRE.exec(nazwiskoTxt) != nazwiskoTxt){
            alert("Nieprawidłowe dane!");
            return;
        }
        form1.submit();
    }
</script>
```

Sposób tworzenia tych wyrażeń jest z pewnością jasny dla każdego, kto zapoznał się z dotychczasowym materiałem bieżącej lekcji. Zwróćmy uwagę, że ostatnie z wyrażeń pozwala jedynie na stosowanie dużych i małych liter alfabetu łacińskiego, nie będzie więc można skorzystać np. z polskich znaków. Warto zastanowić się, jak poprawić to niedopatrzenie (ćwiczenie 19.4 na końcu lekcji).

Po utworzeniu wyrażeń pobierane jest odwołanie do formularza `form1`, a następnie odczytywane są wartości zawarte w poszczególnych polach tekstowych. Dane te są zapisywane w zmiennych `imięTxt`, `nazwiskoTxt`, `telefonTxt`, `kodTxt`. Dalej, za pomocą instrukcji warunkowej `if` i metody `exec` obiektu `RegExp`, badana jest zgodność odczytanych tekstów z wzorcami zapisanymi w wyrażeniach regularnych. Jeśli wszystkie dane są w odpowiednim formacie, formularz jest wysyłany, jeśli jednak choć jeden tekst nie spełnia wymogów, na ekranie pojawia się informacja o błędnych danych, a wykonywanie funkcji jest przerywane przy użyciu instrukcji `return`.

Zwróćmy jeszcze uwagę na sposób badania zgodności z wzorcem. Otóż, dla każdego tekstu jest wywoływana metoda `exec` odpowiadającego mu obiektu typu `RegExp`. Jak pamiętamy, zwraca ona albo wartość `null`, albo tablicę wynikową, gdzie w pierwszej komórce znajduje się odszukany ciąg. W naszym przypadku chcielibyśmy, aby odszukany ciąg był dokładnie taki, jak przeszukiwany tekst, czyli by wynik działania metody `exec` był taki sam jak ciąg przekazany jej w postaci argumentu. Dla kodu pocztowego osiągamy to za pomocą warunku:

```
kodRE.exec(kodTxt) != kodTxt
```

Czy jednak taka instrukcja na pewno jest poprawna? Rozważmy możliwe sytuacje. Jeśli wzorzec reprezentowany przez `kodRE` nie zostanie znaleziony w tekście `kodTxt`, wynikiem działania metody `exec` będzie wartość `null`. Wartość `null` można porównać z wartością zmiennej `kodTxt`, a więc instrukcja będzie prawidłowa i da wynik `false`. Jeżeli jednak w tekście zawartym w `kodTxt` zostanie odnaleziony wzorzec, rezultatem działania metody `exec` będzie tablica wynikowa (jej postać została opisana w punkcie „Metody używające wyrażeń regularnych”). A zatem nastąpi próba porównania tablicy oraz ciągu znaków. Oczywiście, bezpośrednie porównanie nie jest możliwe (dane będą różnych typów). Dlaczego więc skrypt działa? Dlatego, że w takiej sytuacji nastąpi automatyczna konwersja tablicy do ciągu znaków. Konwersja odbywa się w ten sposób, że tworzony jest ciąg składający się z zawartości wszystkich komórek tablicy oddzielonych od siebie znakami przecinka. W rozpatrywanym przypadku tablica wynikowa będzie miała tylko jedną komórkę zawierającą wyszukany ciąg pasujący do wyrażenia, a więc porównanie będzie możliwe i da właściwy wynik. Warto zastanowić się, co się stanie, jeśli zastosujemy opisaną technikę, a używane przez nas wyrażenia regularne będą zawierały grupowania danych. Czy skrypt zadziała prawidłowo, a jeśli nie, to dlaczego i jak można go poprawić (ćwiczenie 19.5)?

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 19.1.

Popraw wyrażenie `/a.+e/i` z podpunktu „Zachłanność wyrażeń”, tak by dopasowywało możliwie najkrótszy ciąg występujący między literami a i e. Nie używaj znaku ? tworzącego niezachłanną wersję wyrażenia.

Ćwiczenie 19.2.

Popraw skrypt z listingów 5.21 i 5.22 tak, by w przypadku wprowadzenia niepoprawnego wyrażenia regularnego nie występował błąd, ale użytkownik witryny był informowany o tym stosownym komunikatem.

Ćwiczenie 19.3.

W oparciu o kod testujący wyrażenia regularne za pomocą metody exec napisz skrypt pozwalający na testowanie zamiany w tekście wzorców za pomocą metody replace obiektu String.

Ćwiczenie 19.4.

Popraw kod z listingu 5.25 tak, aby użytkownik był informowany, które pola są błędnie wypełnione, oraz by w polach *imię* i *nazwisko* było możliwe wprowadzanie polskich znaków w standardzie Unicode (dla uproszczenia możesz użyć całego zakresu Unicode dla standardów Latin i Latin-A, kody od \u00C0 do \u017F).

Ćwiczenie 19.5.

Zastanów się, co się stanie, jeśli w przykładzie z listingów 5.24 i 5.25 zostaną użyte wyrażenia regularne zawierające grupowania. Jakie poprawki należy wprowadzić, aby i w takiej sytuacji skrypt działał prawidłowo? Zastosuj np. wyrażenia: /\+\d{2}(\(\d{2}\))\ \d{6}/, /(\d{2})-(\d{3})/.

Lekcja 20. Cookies

Cookies są to niewielkie porcje danych tekstowych, które mogą być przesyłane między serwerem a przeglądarką. Przeglądarka przechowuje te dane przez określony czas. Wokół cookies narosło wiele mitów i nieporozumień, z których najpopularniejsze i najmniej prawdziwe jest twierdzenie, że są to programy umieszczane przez serwer na komputerze użytkownika. Nic takiego nie ma miejsca — cookies to zwykłe ciągi znaków. Tylko tyle i aż tyle. Jedyne zagrożenie, jakie się z nimi wiąże, to możliwość śledzenia zachowań użytkownika, przez co pewne specyficzne rodzaje cookies (wysypane przez niektóre strony WWW) są określone (raczej niesłusznie i wyłącznie ze względu na efekt marketingowy) jako spyware przez oprogramowanie wykrywające różnorakie zagrożenia. W lekcji 20. dowiemy się, jak posługiwać się cookies, jakie informacje mogą one przekazywać, jak je zapisywać, odczytywać i ustalać czas ich ważności.

Jak działają cookies?

Jak zaznaczono na wstępie, cookies to zwykłe porcje danych tekstowych. Gdy odwiedzamy jakąś stronę WWW, serwer może wysłać do przeglądarki tzw. nagłówek HTTP zawierający cookies. Pomijając jednak szczegóły techniczne, najczęściej mówimy po prostu, że serwer wysyła cookies do przeglądarki. Przeglądarka zapamiętuje je i przechowuje przez określony czas, który jest czasem ważności danego cookie określonym przez serwer, ale — uwaga — przeglądarka nie musi się do niego stosować. Z reguły można ją ustawić tak, aby np. kasowała wszystkie cookies przy jej zamykaniu. Gdy przeglądarka zapamięta cookies przypisane danej stronie WWW, odsyła je przy każdym kolejnym połączeniu z daną stroną. To bardzo użyteczne. Mechanizm ten wykorzystuje się do najrozmaitszych zastosowań. Przy jego użyciu można np. zapamiętywać ustawienia strony specyficzne dla danego użytkownika, dokonywać uwierzytelniania (weryfikacji użytkowników), czy realizować tzw. koszyk zakupów (w praktyce z cookies korzysta każdy sklep internetowy).

Z czego składa się cookie?

Pojedyncze cookie to ciąg znaków określający nazwę i przypisaną jej wartość oraz dodatkowe parametry. Taki ciąg ma schematyczną postać:

`nazwa=wartość;expires=data;path=ścieżka;domain=domena;secure;`

Oto znaczenie poszczególnych części.

- ◆ *nazwa* — nazwa identyfikująca cookie.
- ◆ *wartość* — wartość danego cookie.
- ◆ *data* — określana również jako *czas ważności* lub *czas życia*. Jest to określenie, kiedy wygasza ważność cookie. Po tym czasie zostanie ono usunięte przez przeglądarkę,
- ◆ *ścieżka* — ścieżka dostępu na serwerze, do której zostanie ograniczona dostępność cookie. Przykładowo ustawienie tego parametru na / powoduje, że cookie dostępne jest w całej domenie, a ustawienie /www/example/, że będzie dostępne jedynie przy odwołaniu do katalogu /www/example/ i katalogów podrzędnych w stosunku do /www/example/.
- ◆ *domena* — określa domenę, w której cookie będzie dostępne. Przykładowo ustawienie na .domain.com powoduje, że cookie będzie dostępne we wszystkich poddomenach domeny domain.com, natomiast ustawienie na www.domain.com powoduje, że będzie dostępne jedynie w obrębie domeny www.domain.com.
- ◆ *secure* — obecność tego parametru powoduje, że cookie będzie przesyłane jedynie przez bezpieczne połączenia HTTPS; jeśli parametr zostanie pominięty, cookie będzie przesyłane również przez zwykłe połączenia HTTP.

Obligatoryjne jest jedynie użycie ciągu nazwa=wartość określającego nazwę danego cookie i przypisaną mu wartość. Przykładem może być ciąg:

`imie=Jan`

Tak określone cookie będzie miało czas życia (ważności) ograniczony do jednej sesji przeglądarki, a to oznacza, że zostanie skasowane po jej zamknięciu. Jeśli chcemy, aby było przechowywane dłużej, musimy użyć parametru określającego ten czas — `expires` — który powinien zawierać datę w formacie:

`Wdy, DD-Mon-YYYY HH:MM:SS GMT`

gdzie:

`Wdy` to trzyliterowy skrót określający dzień tygodnia, np. Wed, Thu itd.;

`DD` to numer dnia w miesiącu w formacie dwucyfrowym, np. 01, 10, 22;

`Mon` to trzyliterowy skrót określający miesiąc, np. Jan, Aug, Dec itd;

`YYYY` to czterocyfrowe określenie roku, np. 2010, 2059 itd.;

`HH` to godzina;

`MM` to minuta;

`SS` to sekunda.

Należy używać czasu GMT (*Greenwich Mean Time*). Zatem poprawne będą określenia:

`Wed, 15-12-2010 12:24:59 GMT`

`Sat, 24-03-2012 18:00:00 GMT`

Tak więc cookie o nazwie `imie` przechowujące wartość `Jan`, ważne do poniedziałku, 14 lipca 2011, do godziny 14:50:00 (czasu GMT; obowiązujący w Polsce czas śródkowieuropejski jest przesunięty o jedną godzinę do przodu) będzie miało postać:

`imie=Jan;expires=Mon, 14-07-2011 14:50:00 GMT`:

Taki ciąg nie musi być tworzony ręcznie — byłoby to wyjątkowo niewygodne. Zazwyczaj używa się w tym celu obiektu `Date` i odpowiednich metod konwertujących dane. Opisaną postać parametru `expires` można uzyskać za pomocą metody `toGMTString`. Obiektem `Date` zajmiemy się jednak dopiero w rozdziale 6., w lekcji 21.

W protokole HTTP w wersji 1.0 parametr `expires` był jedyną możliwością określenia czasu ważności cookies. W wersji 1.1 został jednak dodany inny — `max-age`. Jest prostszy w użyciu, gdyż pozwala na określenie czasu życia w sekundach. Ponieważ obecnie praktyczne każdy serwer i przeglądarka obsługują HTTP 1.1, z powodzeniem można z tego parametru korzystać. Jeśli więc cookie ma być ważne przez 2 godziny, napiszemy:

`max-age=7200`:

A jeśli mają to być dwie doby (48 godzin):

`max-age=172800`:

Użycie parametru `path` powoduje ograniczenie działania cookie do danej ścieżki. Jeśli ten parametr nie zostanie użyty lub będzie miał postać:

/

np.:

imie=Jan;expires=Mon, 14-07-2011 14:50:00 GMT;path=/

cookie będzie dostępne w każdej ścieżce, czyli poniższych i im podobnych:

http://mojadomena.com/

http://mojadomena.com/www/

http://mojadomena.com/pliki/grafika/

Jeżeli jednak parametr ten będzie zawierał konkretną ścieżkę, np.:

path=/dane/pliki/

cookie będzie przesyłane jedynie przy odwołaniu:

http://mojadomena.com/dane/pliki/

W podobny sposób działa parametr domain — ogranicza dostęp do konkretnych poddomen. Jeżeli nie jest obecny lub ma postać:

.nazwadomeny

cookie jest przesyłane przy odwołaniu do każdej poddomeny. A zatem parametr o postaci:

domain=.mojadomena.com

będzie pasował do odwołań, takich jak poniższe i podobnych:

http://www.mojadomena.com

http://download.mojadomena.com

Jeżeli jednak przyjmie postać:

domain=serwis.mojadomena.com

cookie będzie przesyłane tylko przy odwołaniu:

http://serwis.mojadomena.com

A zatem przykładowe cookie:

imie=Jan;expires=Mon, 14-07-2011 14:50:00

GMT;path=/download;/domain=pliki.mojadomena.com;

będzie przechowywało ciąg znaków Jan, do poniedziałku 14 lipca 2011, do godziny 14:50 i będzie przesyłane tylko po odwołaniu do witryny znajdującej się pod adresem <http://pliki.mojadomena.com/download/>.

Gdy korzystamy z cookies, powinniśmy tworzyć je tak, aby zajmowały możliwie najmniej miejsca. Każde cookie może też mieć co najwyżej 4 KB, a w przypadku Internet Explorera w objętości 4 KB muszą się zmieścić wszystkie cookie pochodzące z jednej domeny. Istnieje również ograniczenie dotyczące liczby cookies pochodzących z jednej domeny. Dla Firefoksa i Internet Explorera w wersjach od 14 września 2007 jest to 50, dla Internet Explorera w wersjach sprzed 14 września 2007 — 20, a dla Opery — 30.

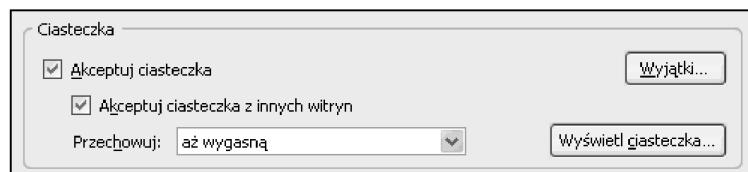
Podglądarki cookies w przeglądarkach

Część przeglądarek udostępnia możliwość podejrzenia zebranych przez nie cookies. W Firefoksie należy z menu *Narzędzia* wybrać pozycję *Opcje*, a następnie kliknąć zakładkę *Prywatność*. Na karcie widocznej na rysunku 5.12 w sekcji *Ciasteczka* można wtedy zdecydować, czy przeglądarka ma przyjmować cookies i jak długo je przechowywać. Kliknięcie przycisku *Wyświetl ciasteczka* spowoduje z kolei wyświetlenie zebranych ciasteczek z podziałem na domeny, co zostało pokazane na rysunku 5.13.

Można przeglądać zawartość cookies, a także usuwać wybrane z nich.

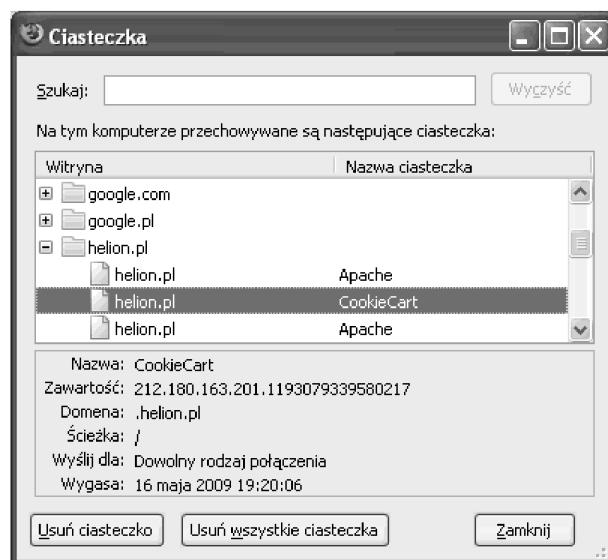
Rysunek 5.12.

Sekcja Ciasteczka
zakładki Prywatność
w przeglądarce
Firefoks



Rysunek 5.13.

Menedżer cookies
przeglądarki Firefox



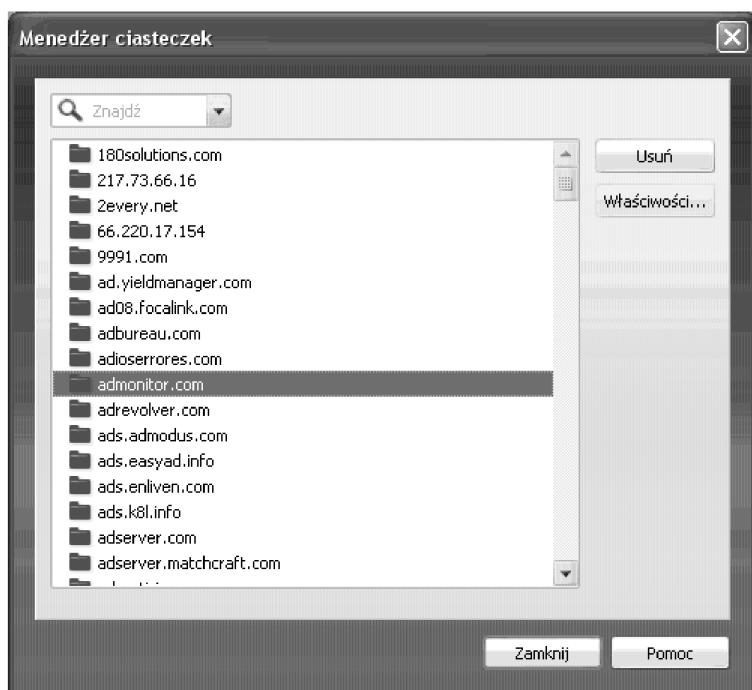
W przeglądarce Opera dostęp do menedżera cookies osiągniemy przez wybranie z menu *Narzędzia* opcji *Zaawansowane*, a następnie *Ciasteczka*. Po wykonaniu tych czynności na ekranie zobaczymy okno *Menedżer ciasteczek*. Jego wygląd został przedstawiony na rysunku 5.14.

Cookies i JavaScript

Na zapis i odczyt cookies pozwala właściwość *cookie* obiektu *document* (tabela 4.4 w lekcji 12.). Zawiera ona ciąg znaków opisujący cookies przypisane danej witrynie. Aby zapisać cookie, należy użyć konstrukcji o schematycznej postaci:

```
document.cookie = "nazwa=wartość;expires=data;path=ścieżka;domain=domena;secure;" ;
```

Rysunek 5.14.
Menedżer cookies
przeglądarki Opera



Można, oczywiście, pominąć parametry opcjonalne, np.:

```
document.cookie = "nazwa=wartość";
document.cookie = "nazwa=wartość;expires=data";
```

Jeżeli zatem w skrypcie znajdzie się przykładowy wiersz:

```
document.cookie = "imie=Jan;max-age=7200";
```

spowoduje on zapisanie przez przeglądarkę cookie o nazwie imie i wartości Jan, które będzie ważne przez 2 godziny. Jeżeli chcemy zapisać kilka cookies, robimy to za pomocą osobnych instrukcji przypisania, np. dwa następujące wiersze spowodują zapisanie dwóch cookies:

```
document.cookie = "imie=Jan;max-age=7200";
document.cookie = "nazwisko=Kowalski;max-age=7200";
```

Odczyt wygląda nieco inaczej. Jeżeli bowiem pobierzemy zawartość właściwości cookie, tym samym odczytamy zawartość wszystkich cookies przypisanych witrynie. Uwaga: odczytany ciąg będzie miał postać:

```
nazwa1=wartość1; nazwa2=wartość2; ...nazwaN=wartość
```

Będzie to zatem zestaw par nazwa-wartość oddzielonych od siebie znakami średnika i spacji. Każda para będzie opisywała jedno cookie. Wartości parametrów (expires, path itd.) nie będą uwzględnione. Jeżeli witryna nie będzie zawierała żadnego cookie, we właściwości cookie obiektu document znajdował się będzie pusty ciąg znaków. Takie zachowanie oznacza, że w celu pobrania wartości konkretnych cookies musimy dokonać analizy ciągu i pobrać podciąg odpowiadający danej wartości.

Wykonajmy konkretny przykład pokazujący, jak zapisywać i odczytywać cookies. Będzie działał w sposób następujący: gdy na stronie nie ma cookies, zapisze dwa, zawierające losowe liczby, o czasie życia równym 10 sekund, w przeciwnym przypadku — odczyta wartości zapisane w cookies i wyświetli je na stronie. Tak krótki czas ważności pozwoli na swobodne testowanie działania skryptu. Kod HTML będzie miał postać, jaką stosowaliśmy w 2. rozdziale (listing 2.1). Kod JavaScript realizujący pierwszą część przedstawionego zadania został przedstawiony na listingu 5.26.

Listing 5.26. Część skryptu zapisująca cookies

```
var str = "";
if(document.cookie == ""){
    var liczba1 = Math.random();
    var liczba2 = Math.random();

    var cookiStr = "";
    cookiStr = "liczba1=" + liczba1;
    document.cookie = cookiStr + ";max-age=10";
    cookiStr = "liczba2=" + liczba2;
    document.cookie = cookiStr + ";max-age=10";
    str += "Nie odnaleziono cookies. Zostały wygenerowane";
    str += "nowe wartości: " + liczba1;
    str += " i " + liczba2 + ".";
}
```

Najpierw badamy, czy na stronie są już cookies. Robimy to przez porównanie wartości właściwości cookie z pustym ciągiem znaków:

```
if(document.cookie == ""){
```

Jeśli równość występuje (czyli właściwość cookie zawiera pusty串 znaków), wykonywane są instrukcje zapisujące cookies w przeglądarce. Najpierw za pomocą metody random obiektu Math generowane są dwie losowe wartości, które są zapisywane w pomocniczych zmiennych liczba1 i liczba2. Następnie generowany jest串 opisujący pierwsze cookie. Ciąg ten ma schematyczną postać:

```
liczba1=wartość1:max-age=10
```

Serializowany cookie jest przypisywany właściwości cookie obiektu document. Tym samym, ustawiane jest pierwsze cookie. Oczywiście, w miejscu oznaczonym wartością wprowadzana jest wartość przypisana zmiennej liczba1. Następnie wykonywane są analogiczne czynności zapisujące drugie cookie. Ma ono schematyczną postać:

```
liczba2=wartość2:max-age=10
```

Na zakończenie przygotowywany jest komunikat informujący o tym, że nie znaleziono cookies, a zatem zostały wygenerowane dwie nowe losowe wartości. Komunikat jest przypisywany zmiennej str dzięki czemu pojawi się na ekranie.

Nieco bardziej złożone czynności trzeba wykonać, gdy cookies są obecne i konieczne jest odczytanie ich wartości. Ten fragment kodu został przedstawiony na listingu 5.27.

Listing 5.27. Część skryptu odczytująca cookies

```
else{
    var cookieStr = document.cookie;
    var start; var end;
    if((start = cookieStr.indexOf("liczba1")) != -1){
        end = cookieStr.indexOf(":", start + 8);
        if(end == -1) end = cookieStr.length;
        var wartość = cookieStr.substring(start + 8, end);
        str += "Znaleziono cookie odpowiadające pierwszej liczbie. ";
        str += "Odczytana wartość to " + wartość + ". ";
    }
    else{
        str += "Pierwsza wartość nie mogła być odczytana. ";
    }
    if((start = cookieStr.indexOf("liczba2")) != -1){
        end = cookieStr.indexOf(":", start + 8);
        if(end == -1) end = cookieStr.length;
        var wartość = cookieStr.substring(start + 8, end);
        str += "Znaleziono cookie odpowiadające drugiej liczbie. ";
        str += "Odczytana wartość to " + wartość + ". ";
    }
    else{
        str += "Druga wartość nie mogła być odczytana."
    }
}
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Najpierw ciąg opisujący cookies jest przypisywany pomocniczej zmiennej cookieStr i deklarowane są zmienne start oraz end. Będą one wyznaczały część ciągu (indeksy początkowego i końcowego znaku), który zawiera poszukiwaną wartość. Wiemy, że ciąg ten powinien mieć postać:

liczba1=wartość1; liczba2=wartość2

Najpierw chcemy odczytać wartość przypisaną cookie o nazwie liczba1. Sprawdzamy zatem, czy w ciągu cookieStr znajduje się ciąg liczba1=, jeśli tak jest, zapisujemy w zmiennej start indeks tego wystąpienia i wykonujemy dalsze czynności. Za wykonanie tej operacji odpowiada złożona instrukcja:

```
if((start = cookieStr.indexOf("liczba1")) != -1){
```

Najpierw wywołuje ona metodę indexOf obiektu cookieStr, czyli sprawdza, czy w ciągu reprezentowanym przez cookieStr znajduje się ciąg liczba1=. Wynik działania tej metody zostaje przypisany zmiennej start, a następnie instrukcja bada, czy wartość zwrocona przez metodę jest różna od -1. Jeśli jest różna, oznacza to, że ciąg liczba1= został odnaleziony, są więc wykonywane instrukcje bloku if. Jeśli jest równa -1, oznacza to, że ciąg nie został odnaleziony, a więc nie ma cookie o nazwie liczba1 (co sugerowałoby błąd w działaniu skryptu). W takiej sytuacji jest wykonywany blok else, który do zmiennej str dopisuje komunikat o problemie z odczytem pierwszej wartości.

W bloku `if` musimy wyodrębnić wartość przypisaną cookie `liczba1`, czyli to, co zaczyna się za ciągiem `liczba1=` i kończy się znakiem średnika lub znakiem końca wiersza. Badamy zatem, czy począwszy od indeksu określonego przez `start + 8` (ciąg `liczba1=` ma osiem znaków) występuje znak średnika:

```
end = cookieStr.indexOf(":", start + 8);
```

czyli czy zmienna `end` ma wartość różną od `-1`. Jeśli nie występuje (`end = -1`), oznacza to, że wartość przypisana cookie `liczba1` kończy się wraz z końcem ciągu `cookieStr`. Wtedy zmiennej `end` przypisywana jest długość ciągu `cookieStr`, czyli indeks ostatniego znaku `+1`:

```
if(end == -1) end = cookieStr.length;
```

Jeżeli jednak znak średnika występuje, czyli zmienna `end` ma wartość różną od `-1`, nie ma potrzeby wykonywania innych przypisań, gdyż oznacza to, że od razu wyznacza ona indeks końca ciągu będącego wartością cookie `liczba1`.

Po wykonaniu opisanych czynności zmienna `start` będzie zawierała indeks znaku, od którego rozpoczyna się wartość ciągu `liczba1=`, a zmienna `end` — indeks znaku, w którym kończy się poszukiwana przez nas wartość. W związku z tym, aby ją uzyskać, wystarczy użyć metody `substring` w postaci:

```
var wartość = cookieStr.substring(start + 8, end);
```

Do `start` dodajemy `8`, bowiem tyle właśnie znaków ma ciąg `liczba1=`. Uzyskana wartość cookie o nazwie `liczba1` jest przypisywana zmiennej pomocniczej `wartość`, która następnie używana jest do skonstruowania ciągu zawierającego komunikat, jaki ma się pojawić na ekranie.

Po zakończeniu opisanych procedur wykonywane są czynności analogiczne do opisanych, które służą pobraniu zawartości cookie o nazwie `liczba2`.

A więc przy pierwszym wczytaniu skryptu do przeglądarki ukaże się widok zaprezentowany na rysunku 5.15. Ponieważ nie było żadnych cookies, zostały wygenerowane dwie losowe wartości, pierwsza z nich została zapisana w cookie o nazwie `liczba1`, a druga — w cookie o nazwie `liczba2`.

Rysunek 5.15.

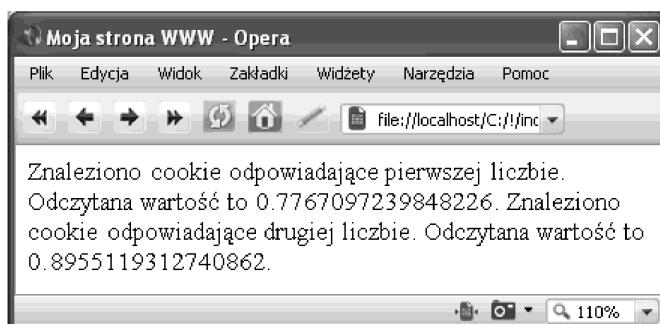
Cookies nie zostały odnalezione



Jeśli teraz w ciągu 10 sekund ponownie odwiedzimy stronę (odświeżymy ją), zobaczymy widok przedstawiony na rysunku 5.16. Przeglądarka zachowała cookies, które powstały przy pierwszych odwiedzinach, a więc ich wartości zostały odczytane i wyświetcone na witrynie. Po upływie 10 sekund od pierwszego wejścia na witrynę cookies zostaną skasowane, a więc kolejne jej odświeżenie spowoduje powtórzenie całej procedury.

Rysunek 5.16.

Dane zostały odczytane z cookies



Zwróćmy uwagę, że zastosowany sposób odczytu cookies umożliwił, co prawda, pobranie danych, jednak był bardzo mało elastyczny. Co więcej, kod odczytujący pierwsze cookie w dużej mierze powtórzył się przy odczytaniu drugiego. Zmieniała się tylko odszukiwana w ciągu nazwa. Może więc warto napisać jedną uniwersalną funkcję odczytującą wartość cookie o zadanej nazwie? Wydaje się, że to bardzo dobry pomysł. Funkcja taka mogłaby przyjąć postać zaprezentowaną na listingu 5.28.

Listing 5.28. Funkcja odczytująca cookie o zadanej nazwie

```
function getCookie(nazwa)
{
    var cookieStr = document.cookie;
    var start; var end;
    if((start = cookieStr.indexOf(nazwa + "=")) == -1)
        return false;
    start += nazwa.length + 1;
    if((end = cookieStr.indexOf(";", start)) == -1)
        end = cookieStr.length;
    return cookieStr.substring(start, end);
}
```

Funkcja przyjmuje argument określający nazwę cookie, którego wartość chcemy pobrać, a działa w sposób podobny do opisanego wyżej. Najpierw ciąg opisujący wszystkie cookies jest zapisywany w pomocniczej zmiennej cookieStr i deklarowane są zmienne start oraz end. Następnie odczytywany jest indeks wystąpienia ciągu znajdującego się w argumencie nazwa uzupełnionego o znak =. Indeks wystąpienia takiego ciągu jest zapisywany w zmiennej start. Jeżeli wartością start jest -1 (czyli ciąg nie został znaleziony), oznacza to, że nie ma cookie o nazwie przekazanej w postaci argumentu i jest zwracana wartość false:

```
if((start = cookieStr.indexOf(nazwa + "=")) == -1)
    return false;
```

Tym samym funkcja kończy działanie. W przeciwnym przypadku wartość zapisana w start jest powiększana tak, by wskazywała pierwszy znak wartości cookie:

```
start += nazwa.length + 1;
```

Następnie odszukiwany jest znak : oznaczający koniec wartości cookie, a indeks jego wystąpienia jest zapisywany w zmiennej end. Gdy okaże się, że zmiennej jest równa -1, będzie to oznaczało, że wartość danego cookie kończy się wraz z ciągiem znaków zapisanych w cookieStr. Dlatego w takiej sytuacji w end zapisujemy po prostu długość ciągu cookieStr.

```
if((end = cookieStr.indexOf(":", start)) == -1)
    end = cookieStr.length;
```

Ostatecznie, po wykonaniu wszystkich opisanych czynności zmienna start będzie wskazywać początek wartości cookie, a end — jej koniec. A więc wartość ta może być pobrana za pomocą metody substring i zwrócona jako efekt działania funkcji, przy użyciu instrukcji return:

```
return cookieStr.substring(start, end);
```

Rozważmy, jak będzie działać funkcja, jeżeli właściwość document.cookie będzie zawierała ciąg:

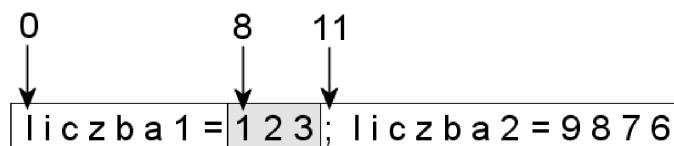
```
liczba1=123; liczba2=9876
```

a poszukiwać będziemy wartości cookie o nazwie liczba1, czyli funkcja zostanie wywołana np. jako:

```
getCookie("liczba1");
```

Najpierw poszukiwany będzie ciąg liczba1=. Zostanie odnaleziony w indeksie 0 (cały ciąg zaczyna się bowiem od podcięgu liczba1=), a więc zmienna start otrzyma taką wartość. Następnie zmienna start zostanie powiększona o długość ciągu liczba1= (start += nazwa.length + 1;), czyli otrzyma wartość 8. Następnie zostanie odszukany znak : (szukanie zacznie się od znaku o indeksie 8 — cookieStr.indexOf(":", start)). Zostanie znalezionej w indeksie 11 i taka wartość zostanie przypisana zmiennej end. Tym samym, indeksy start i end - 1 (to nie błąd, jak pamiętamy drugim argumentem metody substring jest indeks kolejnego znaku za ciągiem, który chcemy wyekstrahować) wyznaczają część ciągu odpowiadającą wartości cookie o nazwie liczba1 (rysunek 5.17).

Rysunek 5.17.
Ekstrakcja wartości cookie o nazwie liczba1



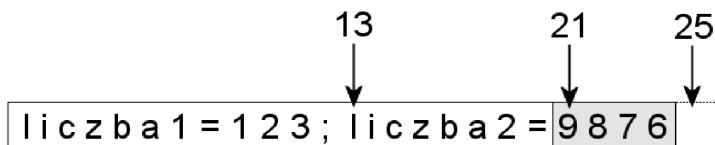
Przy poszukiwaniu wartości cookie o nazwie liczba2 działanie będzie nieco inne. Poszukiwanym ciągiem będzie tym razem liczba2= i zostanie odnaleziony w indeksie 13. Taka też wartość zostanie przypisana zmiennej start. Następnie zostanie ona powiększona o długość ciągu liczba2=, czyli start osiągnie wartość 21. W kolejnym kroku,

począwszy od indeksu wskazywanego przez start, będzie poszukiwany znak :. Nie zostanie odnaleziony (na końcu ciągu opisującego cookies nie ma znaku średnika), a więc zmienna end otrzyma wartość -1. Instrukcja warunkowa:

```
if((end = cookieStr.indexOf(":", start)) == -1)
    end = cookieStr.length;
```

spowoduje, że wartość ta zostanie zamieniona na długość ciągu cookieStr, a więc na 25. Tym samym, indeksy start i end - 1 wyznaczają wartość cookie o nazwie liczba2 (rysunek 5.18).

Rysunek 5.18.
Ekstrakcja wartości
cookie o nazwie liczba2



Inną metodą przetwarzania cookies jest zebranie ich w jednym obiekcie w taki sposób, by właściwości tego obiektu odpowiadały nazwom cookies, a wartości właściwości — wartościom cookies. Jeśli więc document.cookie będzie zawierało ciąg:

liczba1=123; liczba2=9876

to odzwierciedlający cookies obiekt będzie miał dwie właściwości: pierwszą o nazwie liczba1 i wartości 123 oraz drugą o nazwie liczba2 i wartości 9876. Funkcja przetwarzająca ciąg opisujący cookies na taki obiekt została przedstawiona na listingu 5.29.

Listing 5.29. Funkcja tworząca obiekt zawierający wartości cookies

```
function getCookies()
{
    var tab = document.cookie.split(";");
    var obj = new Object();
    for(i = 0; i < tab.length; i++){
        var arr = tab[i].split("=");
        if(arr.length != 2)
            continue;
        obj[arr[0]] = arr[1];
    }
    return obj;
}
```

Ponieważ w ciągu zapisanym w document.cookie poszczególne wartości są od siebie oddzielone znakami średnika i spacji, podział na pojedyncze cookie jest dokonywany za pomocą metody split:

```
var tab = document.cookie.split(";" );
```

W ten sposób otrzymujemy tablicę tab zawierającą ciągi o postaci:

nazwa=*wartość*

odpowiadające pojedynczym cookies. Następnie tworzony jest obiekt obj, w którym zostaną zapisane nazwy i wartości cookies.

Zawartość tablicy tab jest odczytywana w pętli for. Ciąg o powyższej strukturze jest w niej rozbijany na części za pomocą metody split:

```
var arr = tab[i].split("=");
```

W wyniku uzyskiwana jest tablica arr, która w pierwszej komórce zawiera nazwę cookie, a w drugiej — wartość cookie. Za pomocą instrukcji warunkowej if badamy, czy na pewno tablica arr zawiera dwie komórki:

```
if(arr.length != 2)
```

Jeśli nie, oznacza to błąd w strukturze ciągu i jest wykonywana instrukcja continue rozpoczynająca kolejny przebieg pętli. Jeśli tak, wykonywana jest instrukcja tworząca nową właściwość obiektu obj:

```
obj[arr[0]] = arr[1];
```

Po zakończeniu pętli obiekt obj jest zwracany jako wynik działania funkcji get-Cookies:

```
return obj;
```

Zliczanie liczby odwiedzin

Na zakończenie lekcji użyjmy cookies do zliczania liczby odwiedzin na stronie w zadanym okresie, powiedzmy 30 dni. Korzystać będziemy z cookie o nazwie liczba0dw, które będzie przechowywało liczbę wizyt. Skorzystamy też z funkcji getCookie przygotowanej w poprzedniej części lekcji (listing 5.28). Skrypt przyjmie postać zaprezentowaną na listingu 5.30.

Listing 5.30. Skrypt zliczający liczbę odwiedzin na stronie

```
var str = "";
var czasZycia = 60 * 60 * 24 * 30;

function getCookie(nazwa)
{
    //treść funkcji getCookie
}

var liczba0dw;

if((liczba0dw = getCookie("liczba0dw")) === false){
    document.cookie = "liczba0dw=1;max-age=" + czasZycia;
    str += "Na tej stronie nie było Cię w ciągu ostatnich 30 dni.";
}
else{
    if(isNaN(liczba0dw = parseInt(liczba0dw)))
        liczba0dw = 0;
    var cookieStr = "liczba0dw=" + ++liczba0dw + ";";
    cookieStr += "max-age=" + czasZycia;
```

```
document.cookie = cookieStr;  
str += "To Twoje " + liczbaOdw + ". odwiedziny w ciągu ";  
str += "ostatnich 30 dni.;"  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Czas ważności cookie, czyli okres, w którym mają być zliczane wizyty, jest przechowywany w zmiennej `czasŻycia`. Jest ona deklarowana na początku skryptu i używana w dalszej jego części jako wartość parametru `max-age`. Wartość zmiennej wynika z operacji arytmetycznej $60 \times 60 \times 24 \times 30$ określającej liczbę sekund odpowiadającą 30 dniom. Za deklaracją zmiennej została umieszczona treść funkcji `getCookie` (na listingu 5.30 została pominuta, gdyż jest identyczna z tą z listingu 5.28), a dalej deklaracja zmiennej `liczbaOdw`, która odzwierciedla dotyczącą liczbę odwiedzin. Wartość tej zmiennej jest pobierana przez wywołanie funkcji `getCookie`, co odbywa się w złożonej instrukcji warunkowej:

```
if((liczbaOdw = getCookie("liczbaOdw")) === false){
```

Technikę tę stosowaliśmy już w przykładach z poprzedniej części lekcji. Po wykonaniu tej instrukcji zmienna `liczbaOdw` będzie zawierała wartość zapisaną w cookie o takiej samej nazwie (o ile takie cookie było obecne w przeglądarce) lub też wartość `false` (o ile cookie `liczbaOdw` nie było).

Najpierw rozpatrywany jest przypadek drugi (blok `if`). Zakładamy wtedy, że są to pierwsze odwiedziny, trzeba więc zapisać cookie w przeglądarce:

```
document.cookie = "liczbaOdw=1;max-age=" + czasŻycia;
```

Przypadek pierwszy wymaga użycia większej liczby instrukcji (blok `else`). Przede wszystkim odczytane dane są ciągiem znaków. Trzeba je przekształcić na wartość liczbową. Co więcej, nie możemy założyć, że na pewno w cookie jest wartość liczbową, może się okazać, że jest to inny ciąg (powstały choćby w wyniku jakiegoś błędu). Dlatego też korzystamy z instrukcji:

```
if(isNaN(liczbaOdw = parseInt(liczbaOdw)))
```

Najpierw przypisuje ona zmiennej `liczbaOdw` wynik wywołania funkcji `parseInt` (`liczbaOdw`), a więc wynik konwersji wartości zapisanej w `liczbaOdw` na wartość całkowitą. Jeśli operacja ta zakończy się powodzeniem, jest koniec działania tej instrukcji. W przeciwnym przypadku wartością zwróconą przez `parseInt` będzie `NaN`. Tym samym, wykonana zostanie występująca po `if` instrukcja:

```
liczbaOdw = 0;
```

przypisująca zmiennej `liczbaOdw` wartość 0. A więc gdy stwierdzimy, że wartość odczytana z cookie jest błędna, zachowanie skryptu będzie podobne jak w przypadku pierwszych odwiedzin.

Gdy już ostatecznie zostanie ustalona wstępna wartość zmiennej `liczba0dw`, tworzone jest cookie, w którym jej wartość jest zwiększana o jeden:

```
var cookieStr = "liczba0dw=" + ++liczba0dw + ":";
```

i to cookie jest zapisywane w przeglądarce:

```
document.cookie = cookieStr;
```

A więc każde kolejne odwiedziny będą powodowały zwiększenie wartości zapisanej w cookie o 1.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 20.1.

Zmodyfikuj kod przykładu z listingów 5.26 i 5.27 tak, by do odczytu cookies używana była funkcja z listingu 5.28.

Ćwiczenie 20.2.

Popraw kod przykładu z listingu 5.26 i 5.27 tak, by do odczytu cookies korzystał z metody `getCookies` z listingu 5.29 oraz by w przypadku nieodnalezienia tylko jednego z cookies było ono generowane.

Ćwiczenie 20.3.

Napisz skrypt, który umożliwi użytkownikowi wybór ulubionego koloru tła strony i zapamiętaj ten wybór w cookie. Przy każdych kolejnych odwiedzinach tego użytkownika automatycznie zmieniaj kolor strony na wybrany przez niego.

Ćwiczenie 20.4.

Zmodyfikuj skrypt z listingu 5.30 tak, by liczba dni, w których zliczane są odwiedziny, była określana za pomocą zmiennej globalnej i automatycznie uwzględniana w komunikatach wyświetlanych na stronie.

Rozdział 6.

Data i czas

Lekcja 21. Obsługa daty i czasu

W trakcie pisania skryptów nierzadko będziemy chcieli wykonać operacje na dacie i (lub) czasie. Typową operacją jest np. pobranie aktualnej daty czy bieżącego czasu (godziny, minuty, sekundy). W JavaScriptu na manipulację takimi danymi pozwalają obiekty typu Date. Właśnie tym typem danych zajmiemy się w 21. lekcji. Poznamy różne typy konstruktorów, a tym samym, możliwości tworzenia obiektów opisujących datę i czas. Zaznajomimy się też z wszystkimi przydatnymi metodami, a także sposobami pobierania danych, korzystania z nich oraz używania uzyskanych informacji w praktyce.

Obiekt Date

Obiekty typu Date służą do reprezentacji daty i czasu. Mogą być tworzone poprzez wywołanie jednego z poniższych konstruktorów:

```
Date();
Date(milisekundy);
Date(dateStr);
Date(rok, miesiąc, dzień[, godzina[, minuta[, sekunda[, milisekunda]]]]);
```

W pierwszym przypadku zostanie utworzony obiekt reprezentujący bieżącą datę i czas, np.:

```
var data = new Date();
```

„Bieżącą” w tym przypadku oznacza aktualną w momencie tworzenia obiektu. Po wywołaniu konstruktora parametry takiego obiektu pozostają niezmienne i nie są w żaden sposób aktualniane.

Druga postać konstruktora pozwala na użycie parametru *milisekundy*, który określa liczbę milisekund, jakie upłyнуły od godziny 00:00:00.000, 1 stycznia 1970 roku do daty, jaką chcemy uzyskać. Parametr *milisekundy* powinien być wartością całkowito-liczbową. Przykładowe wywołanie może mieć postać:

```
var data = new Date(3600000);
```

Trzecia postać konstruktora pozwala na użycie ciągu opisującego datę i czas. Ciąg ten powinien mieć format zgodny ze strukturą podaną przy opisie metody *parse* w dalszej części lekeji, np.:

```
var data = new Date("Sat, 17 Dec 2001 14:02:58 GMT+0100");
```

Czwarta postać przyjmuje zestaw parametrów, których znaczenie jest zgodne z nazwami. Wszystkie parametry powinny być wartościami typu *int*. Parametry ujęte w nawiasy kwadratowe są opcjonalne i mogą być pomijane od strony prawej do lewej, np.:

```
var data1 = new Date(2010, 7, 14, 23, 18, 5);
var data2 = new Date(2012, 10, 22);
```

Obiekty typu *Date* udostępniają zestaw metod zebranych w tabeli 6.1. Większość z nich ma również odpowiedniki operujące na czasie UTC (*Universal Time Coordinated*)¹. Metody te nie zostały ujęte w tabeli, gdyż ich działanie jest tożsame z wymienionymi w niej, a jedyną różnicą jest to, że nie działają na czasie lokalnym, ale na czasie UTC. Te metody to: *getUTCDate*, *getUTCDay*, *getUTCFullYear*, *getUTCHours*, *getUTCMilliseconds*, *getUTCMilliseconds*, *getUTCSeconds*, *setUTCDate*, *setUTCFullYear*, *setUTCHours*, *setUTCMilliseconds*, *setUTCMilliseconds*, *setUTCMonth*, *setUTCSeconds*.

Tabela 6.1. Metody obiektu *Date*

Metoda	Wywołanie	Opis	Dostępność
<i>getDate</i>	<i>data.getDate()</i>	Zwraca dzień miesiąca.	od JavaScript 1.0 i JScript 1.0
<i>getDay</i>	<i>data.getDay()</i>	Zwraca wartość od 0 do 7, reprezentującą dzień tygodnia, 0 — niedziela, 1 — poniedziałek, 2 — wtorek itd.	od JavaScript 1.0 i JScript 1.0
<i>getFullYear</i>	<i>data.getFullYear()</i>	Zwraca rok, w postaci czterocyfrowej. Zaleca się stosowanie tej metody zamiast przestarzałej metody <i>getYear</i> .	od JavaScript 1.3 i JScript 3.0
<i>getHours</i>	<i>data.getHours()</i>	Zwraca godzinę czasu reprezentowanego przez obiekt <i>data</i> .	od JavaScript 1.0 i JScript 1.0
<i>getMilliseconds</i>	<i>data.getMilli- seconds()</i>	Zwraca liczbę milisekund czasu reprezentowanego przez obiekt <i>data</i> .	od JavaScript 1.3 i JScript 3.0
<i>getMinutes</i>	<i>data.getMinutes()</i>	Zwraca liczbę minut czasu reprezentowanego przez obiekt <i>data</i> .	od JavaScript 1.0 i JScript 1.0

¹ Pomijając zastosowania specjalistyczne, można przyjąć, że czas UTC jest równoważny czasowi GMT (Greenwich Mean Time) i zachodnioeuropejskiej strefie czasowej.

Tabela 6.1. Metody obiektu Date (*ciąg dalszy*)

Metoda	Wywołanie	Opis	Dostępność
getMonth	<code>data.getMonth()</code>	Zwraca wartość określającą miesiąc daty reprezentowanej przez obiekt <code>data</code> , przy czym 0 oznacza styczeń, 1 — luty itd.	od JavaScript 1.0 i JScript 1.0
getSeconds	<code>data.getSeconds()</code>	Zwraca liczbę sekund czasu reprezentowanego przez obiekt <code>data</code> .	od JavaScript 1.0 i JScript 1.0
getTime	<code>data.getTime()</code>	Zwraca liczbę milisekund pomiędzy 1 stycznia 1970 roku, godziną 0:00:00.000 a datą reprezentowaną przez obiekt <code>data</code> .	od JavaScript 1.0 i JScript 1.0
getTimezoneOffset	<code>data.getTimezoneOffset()</code>	Zwraca liczbę minut będącą różnicą pomiędzy bieżącą strefą czasową a czasem GMT.	od JavaScript 1.0 i JScript 1.0
getYear	<code>data.getYear()</code>	Zwraca rok daty reprezentowanej przez obiekt <code>data</code> . Stosowanie tej metody nie jest zalecane, gdyż wynik jej działania jest odmienny w różnych implementacjach.	od JavaScript 1.0 i JScript 1.0
parse	<code>Date.parse(<i>ciąg</i>)</code>	Jest to metoda statyczna, która może być wywoływana bez tworzenia nowego obiektu typu Date. Przetwarza <i>ciąg</i> na datę i zwraca ją w postaci liczby milisekund, które uplynęły od 1 stycznia 1970 roku, godziny 0:00:00.000.	od JavaScript 1.0 i JScript 1.0
setDate	<code>data.setDate(<i>dzień</i>)</code>	Ustawia dzień miesiąca.	od JavaScript 1.3 i JScript 3.0
setFullYear	<code>data.setFullYear(<i>rok</i> [, <i>miesiąc</i> [, <i>dzień</i>]])</code>	Ustawia rok na wartość przekazaną w postaci parametru <i>rok</i> . Jeżeli zostanie podany opcjonalny argument <i>miesiąc</i> , zostanie zmieniony również miesiąc, jeżeli zostaną podane parametry <i>miesiąc</i> i <i>dzień</i> , zostaną również zmienione miesiąc i dzień daty.	od JavaScript 1.3 i JScript 3.0
setHours	<code>data.setHours(<i>godzina</i> [, <i>minuta</i> [, <i>sekunda</i> [, <i>milisekunda</i>]])</code>	Ustawia godzinę czasu reprezentowanego przez obiekt <code>data</code> . Znaczenie parametrów jest zgodne z ich nazwami. Parametry opcjonalne mogą być stosowane od wersji 1.3 JavaScript i 3.0 JScript.	od JavaScript 1.1 (1.3) i JScript 1.0 (3.0)
setMilliseconds	<code>data.setMilliseconds(<i>milisekundy</i>)</code>	Ustawia liczbę milisekund na wartość przekazaną w postaci parametru <i>milisekundy</i> .	od JavaScript 1.3 i JScript 3.0

Tabela 6.1. Metody obiektu Date (ciąg dalszy)

Metoda	Wywołanie	Opis	Dostępność
setMinutes	<i>data.setMinutes</i> ↳(<i>minuty</i> [, <i>sekundy</i>] ↳[, <i>milisekundy</i>])	Ustawia liczbę minut na wartość przekazaną w postaci parametru <i>minuty</i> . Znaczenie parametrów jest zgodne z ich nazwami. Parametry opcjonalne mogą być stosowane od wersji 1.3 JavaScriptu i 3.0 JScript.	od JavaScript 1.1 (1.3) i JScript 1.0 (3.0)
setMonth	<i>data.setMonth</i> ↳(<i>miesiąc</i> [, <i>dzień</i>])	Ustawia miesiąc na wartość przekazaną w postaci parametru <i>miesiąc</i> . Jeżeli zostanie podany opcjonalny parametr <i>dzień</i> , zostanie również zmieniony dzień miesiąca. Parametr <i>dzień</i> może być stosowany w JavaScriptie od wersji 1.3 i JScript od wersji 3.0.	od JavaScript 1.0 (1.3) i JScript 1.0 (3.0)
setSeconds	<i>data.setSeconds</i> ↳(<i>sekundy</i> ↳[, <i>milisekundy</i>])	Ustawia liczbę sekund na wartość przekazaną w postaci parametru <i>sekundy</i> . Jeżeli zostanie podany opcjonalny parametr <i>milisekundy</i> , zostanie również zmieniona liczba milisekund. Parametr <i>milisekundy</i> może być stosowany w JavaScriptie od wersji 1.3 i JScript od wersji 3.0.	od JavaScript 1.0 (1.3) i JScript 1.0 (3.0)
setTime	<i>data.setTime</i> ↳(<i>wartość</i>)	Ustawia datę i czas zgodnie z parametrem <i>wartość</i> . Parametr ten oznacza liczbę milisekund, które uplynęły od 1 stycznia 1970 roku, godziny 0:00:00.000.	od JavaScript 1.0 i JScript 1.0
setYear	<i>data.setYear(rok)</i>	Ustawia rok na wartość przekazaną w postaci parametru <i>rok</i> . Metoda od wersji JavaScriptu 1.3 i JScript 3.0 została zastąpiona przez <i>setFullYear</i> , zatem korzystanie z niej nie jest zalecane.	od JavaScript 1.0 i JScript 1.0
toDateString	<i>data.toDateString()</i>	Zwraca ciąg znaków opisujący datę i czas wskazywane przez obiekt <i>data</i> . Format zwracanego ciągu jest zależny od implementacji.	od JavaScript 1.5 i JScript 5.5
toGMTString	<i>data.toGMTString()</i>	Zwraca ciąg znaków opisujący datę i czas wskazywane przez obiekt <i>data</i> w formacie GMT, np.: Tue, 25 Dec 2007 19:15:06 GMT. Obecnie zaleca się raczej wykorzystywanie metody <i>toUTCString</i> .	od JavaScript 1.0 i JScript 1.0

Tabela 6.1. Metody obiektu Date (*ciąg dalszy*)

Metoda	Wywołanie	Opis	Dostępność
toLocaleDate- →String	<code>data.toLocaleDate- →String()</code>	Zwraca串 znaków opisujący datę wskazywaną przez obiekt <code>data</code> w formacie wykorzystującym ustawienia lokalne systemu (przeglądarki).	od JavaScript 1.5 i JScript 5.5
toLocaleString	<code>data.toLocale- →String()</code>	Zwraca串 znaków opisujący datę i czas wskazywane przez obiekt <code>data</code> w formacie wykorzystującym ustawienia lokalne systemu (przeglądarki).	od JavaScript 1.0 i JScript 1.0
toLocaleTime- →String	<code>data.toLocaleTime- →String()</code>	Zwraca串 znaków opisujący czas wskazywany przez obiekt <code>data</code> w formacie wykorzystującym ustawienia lokalne systemu (przeglądarki).	od JavaScript 1.5 i JScript 5.5
toString	<code>date.toString()</code>	Zwraca串 znaków opisujący datę i czas wskazywane przez obiekt <code>data</code> . Najczęściej jest to format GMT (UTC), występują drobne różnice, w zależności od implementacji.	od JavaScript 1.1 i JScript 2.0
toTimeString	<code>data.toTimeString()</code>	Zwraca串 znaków opisujący czas wskazywany przez obiekt <code>data</code> . Najczęściej jest to format GMT (UTC), występują drobne różnice, w zależności od implementacji.	od JavaScript 1.5 i JScript 5.5
toUTCString	<code>data.toUTCString()</code>	Zwraca串 znaków opisujący datę i czas wskazywane przez obiekt <code>data</code> , w formacie UTC. Występują drobne różnice, w zależności od implementacji.	od JavaScript 1.3 i JScript 3.0

Pobieranie daty i czasu

Jak widać na końcu tabeli 6.1, istnieje wiele metod pobierających datę i czas. Jest ich tak dużo, gdyż zwracają informacje w różnych formatach. Sprawdzmy, jakie występują między nimi różnice. Na listingu 6.1 jest widoczny skrypt używający metod: `toDateString`, `toGMTString`, `toLocaleDateString`, `toLocaleString`, `toLocaleTimeString`, `toString`, `toTimeString` oraz `toUTCString` i wyświetlający wyniki ich działania w oknie przeglądarki.

Listing 6.1. Różne sposoby podawania informacji o dacie i czasie

```
var str = "";
var data = new Date();

str += "toDateString = " + data.toDateString();
str += "<br />toGMTString = " + data.toGMTString();
str += "<br />toLocaleDateString = " + data.toLocaleDateString();
```

```

str += "<br />toLocaleString = " + data.toLocaleString();
str += "<br />toLocaleTimeString = " + data.toLocaleTimeString();
str += "<br />toString = " + data.toString();
str += "<br />toTimeString = " + data.toTimeString();
str += "<br />toUTCString = " + data.toUTCString();

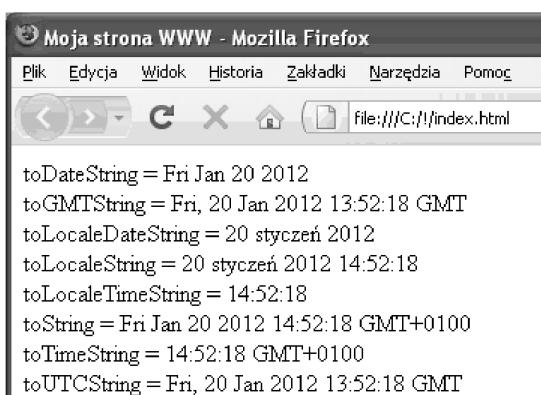
var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;

```

Struktura skryptu jest strukturą stosowaną w rozdziale 2. Skrypt współpracuje z kodem (X)HTML zawierającym warstwę dataDiv. Wszystkie uzyskiwane dane są dopisywane do zmiennej str, a na zakończenie wartość tej zmiennej jest przypisywana właściwości innerHTML obiektu reprezentującego warstwę dataDiv. Po uruchomieniu skryptu łatwo sprawdzimy, jakie różnice występują między użytymi metodami i jakie dane w jakich formatach one zwracają. Jest to widoczne na rysunkach 6.1 (przeglądarka Firefox 3), 6.2 (przeglądarka Internet Explorer 7) i 6.3 (przeglądarka Opera 9).

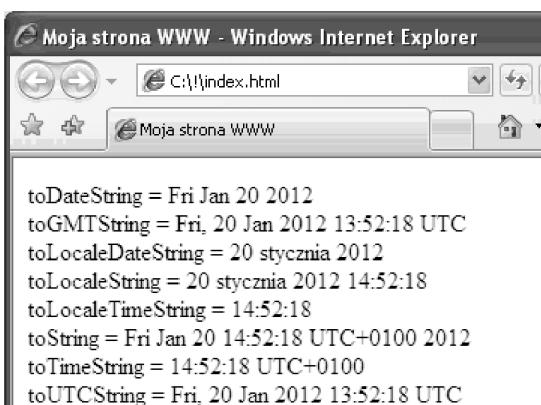
Rysunek 6.1.

Data i czas
w przeglądarce
Firefox



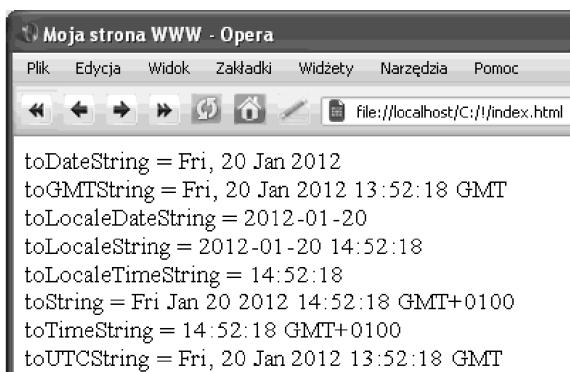
Rysunek 6.2.

Data i czas
w przeglądarce
Internet Explorer



Rysunek 6.3.

Data i czas
w przeglądarce Opera



Jeśli przyjrzymy się dokładnie powyższym rysunkom, zobaczymy, że wyniki działania skryptu wcale nie są identyczne. Okazuje się, że każda przeglądarka nieco inaczej traktuje wyniki działania metod przekształcających datę i czas reprezentowane przez obiekty typu Date na ciągi znaków możliwe do odczytania przez użytkownika witryny. Spójrzmy choćby na działanie metody toLocaleDateString, która powinna podać datę w formacie zgodnym z lokalnymi ustawieniami przeglądarki (systemu). W Firefoxie mamy ciąg:

20 styczeń 2012

Internet Explorer odmienia nazwę miesiąca, podając:

20 stycznia 2012

Z kolei Opera używa wyłącznie zapisu cyfrowego:

2012-01-20

Co w takim razie mamy zrobić, gdy chcemy przedstawiać dane w ścisłe określonym formacie? Trzeba napisać własne procedury przetwarzające dane z obiektów typu Date. Przypuśćmy, że chcielibyśmy uzyskać datę w formacie DD-MM-RRRR (dzień, miesiąc, rok). Możemy wtedy skorzystać ze skryptu zaprezentowanego na listingu 6.2.

Listing 6.2. Uzyskanie daty w określonym formacie

```
var str = "";
var data = new Date();

var dzień = data.getDate();
var miesiąc = data.getMonth() + 1;
var rok = data.getFullYear();

str += ((dzień < 10) ? "0" : "") + dzień;
str += ((miesiąc < 10) ? "-0" : "-") + miesiąc;
str += "-" + rok;

str = "Dziś jest " + str + ".";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Pierwszą czynnością jest utworzenie obiektu typu Date zawierającego bieżącą datę oraz czas i przypisanie go zmiennej date. Następnie za pomocą metod getDate, getMonth i getFullYear jest pobierany aktualny dzień, miesiąc i rok. Ponieważ chcemy, aby dzień i miesiąc zawsze były prezentowane w formacie dwucyfrowym, w przypadku gdy pobrane wartości są mniejsze od 10, dodajemy na ich początek znak 0. Czynności te są wykonywane za pomocą poznanego w lekcji 4. operatora warunkowego. Wykonując dodawanie, unikniemy prezentacji daty w postaci np. 5-7-2011, tylko będzie miała ona postać 05-07-2011. Po uzyskaniu ciąg opisujący datę jest umieszczany wewnątrz komunikatu, zatem po uruchomieniu skryptu zobaczymy widok podobny do zaprezentowanego na rysunku 6.4.

Rysunek 6.4.
*Wyświetlenie daty
w określonym formacie*



Korzystanie z informacji o dacie i czasie

Metody obiektu Date pozwalają na uzyskiwanie różnych przydatnych informacji. Możemy sprawdzić np., jaki jest dzień tygodnia czy miesiąca odpowiadający wybranej dacie. Napiszmy przykładowy skrypt, który po uruchomieniu wyświetli na stronie nazwę aktualnego dnia tygodnia. Tak działający kod jest widoczny na listingu 6.3.

Listing 6.3. Pobranie informacji o dniu tygodnia

```
function pobierzNazwęDnia(data)
{
    var dzień;
    switch(data.getDay()){
        case 0 : dzień = "niedziela"; break;
        case 1 : dzień = "poniedziałek"; break;
        case 2 : dzień = "wtorek"; break;
        case 3 : dzień = "środa"; break;
        case 4 : dzień = "czwartek"; break;
        case 5 : dzień = "piątek"; break;
        case 6 : dzień = "sobota"; break;
        default : dzień = false;
    }
    return dzień;
}

var str = "";
var data = new Date();

var nazwaDnia:
```

```
if((nazwaDnia = pobierzNazwęDnia(data)) !== false){  
    str += "Dziś jest " + nazwaDnia + ":";  
}  
else{  
    str += "Nazwa dnia nie mogła zostać pobrańa.:";  
}  
  
var dataDiv = document.getElementById("dataDiv");  
dataDiv.innerHTML = str;
```

Główne zadanie jest wykonywane przez funkcję pobierzNazwęDnia. Otrzymuje ona w postaci argumentu obiekt typu Date, a zwraca ciąg znaków zawierający nazwę dnia tygodnia odpowiadającego tej dacie. Działanie funkcji rozpoczyna się od deklaracji zmiennej pomocniczej dzień — zostanie jej przypisany ciąg znaków będący rezultatem działania funkcji. Czynnością tą zajmuje się instrukcja switch badająca wartość zwróconą przez wywołanie metody getDay obiektu Date:

```
switch(data.getDay()){
```

Metoda ta zwraca wartość określającą dzień tygodnia: 0 — niedziela, 1 — poniedziałek itd. Zależnie zatem od tego, jaka wartość zostanie zwrócona, wykonana będzie odpowiednia klauzula case, a zmenna dzień otrzyma ciąg znaków opisujący dany dzień, np. dla wtorku zostanie wykonany blok:

```
case 2 : dzień = "wtorek"; break;
```

Na końcu instrukcji switch znajduje się klauzula default:

```
default : dzień = false;
```

Co prawda, jej wykonanie jest mało prawdopodobne (metoda getDay zgodnie z opisem zwraca przecież wyłącznie wartości od 0 do 6), jednak nie zaszkodzi umieszczenie takiego zabezpieczenia na wypadek, gdyby wystąpiło jakieś zachowanie niestandardowe i zwrócona przez metodą wartość okazała się różna od oczekiwanej.

Na zakończenie funkcji pobierzNazwęDnia wartość zapisana w zmiennej dzień jest zwracana za pomocą instrukcji return. Będzie to zatem albo ciąg zawierający nazwę dnia, albo też wartość false w przypadku niespodziewanego zachowania metody getDate. Ta właściwość funkcji jest wykorzystywana w dalszej części skryptu.

Wykonywanie kodu skryptu rozpoczyna się od utworzenia zmiennych str, date i nazwaDnia. Zmiennej data jest przypisywany nowy obiekt typu Date. Wywołanie:

```
var data = new Date();
```

powoduje, że obiekt ten będzie reprezentował datę bieżącą.

Następnie w złożonej instrukcji:

```
if((nazwaDnia = pobierzNazwęDnia(data)) !== false){
```

zmiennej nazwaDnia jest przypisywany wynik działania funkcji pobierzNazweDnia i wartość ta jest od razu porównywana z false. Gdy wynik jest negatywny, czyli gdy nazwaDnia jest różna od false, zmienna ta zawiera nazwę aktualnego dnia tygodnia, jest więc używana jako część komunikatu informacyjnego:

```
str += "Dziś jest " + nazwaDnia + ".";
```

Wtedy po uruchomieniu skryptu zobaczymy widok zaprezentowany na rysunku 6.5 (zakładamy tu, że skrypt został uruchomiony w środę). Gdyby jednak okazało się, że nazwaDnia zawiera wartość false, zostanie wykonany blok else i na ekranie pojawi się komunikat informujący, że nazwa dnia nie mogła zostać pobrana.

Rysunek 6.5.

Wynik działania skryptu wyświetlającego nazwę dnia tygodnia



Wykonajmy jeszcze inny przykład użycia danych udostępnianych przez obiekt Date. Skoro bowiem mamy dostęp do aktualnej godziny, możemy np. uzależnić wygląd strony od pory dnia, w której odwiedzi ją dany użytkownik. To, jakie elementy będą podlegały zmianom, zależy — oczywiście — od konkretnego przypadku i naszej własnej koncepcji. Powiedzmy, że zmieniać ma się kolor tła. Inny będzie rano, inny po południu, wieczorem czy w nocy. Sposób wykonania takiego zadania umieszczono w skrypcie na listingu 6.4.

Listing 6.4. Tło zależne od pory dnia

```
function pobierzKolorTła(data)
{
    godzina = data.getHours();
    var kolor = "#FFFFFF";
    if(godzina > 22 || godzina < 4)
        kolor = "#AAAAAA";
    else if(godzina >= 4 && godzina < 12)
        kolor = "#BBBBBB";
    else if(godzina >= 12 && godzina < 18)
        kolor = "#CCCCCC";
    else if(godzina >= 18 && godzina <= 22)
        kolor = "#DDDDDD";
    return kolor;
}

var data = new Date();
var kolor = pobierzKolorTła(data);
document.body.style.backgroundColor = kolor;
```

Za uzyskanie koloru odpowiada funkcja pobierzKolorTła, której zasada działania jest nieco podobna do funkcji pobierzNazwęDnia z poprzedniego przykładu. Jako argument przyjmuje ona obiekt określający datę, a zwraca kod koloru w formacie `#RRGGBB`, który może być bezpośrednio użyty do modyfikacji stylu CSS. We wnętrzu funkcji za pomocą metody `getHours` pobierana jest godzina czasu reprezentowanego przez obiekt `data`. Wartość ta jest zapisywana w zmiennej `godzina`. Następnie pomocniczej zmiennej `kolor` jest przypisywany kod `#FFFFFF` oznaczający kolor biały. Wartość ta zostanie zwrocona, jeśli nie uda się dopasować żadnego przedziału czasowego do wartości zawartej w `godzina` (co oznaczałoby błąd w działaniu metody `getHours` lub ustalenie przedziałów czasowych niepowtarzających całego zakresu doby).

Do ustalenia koloru używana jest instrukcja warunkowa `if...else if` przypisująca zmiennej `kolor` kod koloru związany z danym przedziałem czasowym. Dla godzin od 22 do 4 jest to kolor `#AAAAAA`, dla godzin od 4 do 12 — `#BBBBBB`, dla godzin od 12 do 18 — `#CCCCCC`, a dla godzin od 18 do 22 — `#DDDDDD`. Uzyskany kod jest przypisywany zmiennej `kolor` i zwracany po zakończeniu instrukcji warunkowej za pomocą instrukcji `return`.

Po uruchomieniu skryptu najpierw powstanie nowy obiekt typu `Date`. Zostanie on przypisany zmiennej `data`, której użyjemy w wywołaniu funkcji `pobierzKolorTła`. Wynik działania tej funkcji zostanie z kolei przypisany zmiennej `kolor`, której wartość zostanie przypisana właściwości `backgroundColor` obiektu `style` przypisanego sekcji `body`. To spowoduje zmianę koloru tła strony. Tę część skryptu można również zapisać w pojedynczej, aczkolwiek mniej czytelnej instrukcji:

```
document.body.style.backgroundColor = pobierzKolorTła(new Date());
```

Formatowanie daty (metoda `parse`)

Obiekt `Date` udostępnia statyczną metodę `parse`, która przetwarza ciąg znaków opisujący datę na liczbę milisekund, które uplynęły między tą datą a 1 stycznia 1970 roku, godziną 0:00:00. Innymi słowy, zwraca tzw. znacznik czasu Uniksa, który może być dalej przetwarzany przez inne metody. Metoda jest statyczna, tzn. niezależna od istnienia obiektów typu `Date` i należy ją wywoływać jako:

```
Date.parse(ciąg_znaków)
```

Standard ECMAScript nie definiuje dokładnie, jak powinien być sformatowany ciąg będący argumentem tej metody. Określa jedynie, że poprawnie muszą być przetworzone dane, które są zwracane przez metody `toString` i `toUTCString`.

Najczęściej przyjmowane jest, że argument powinien być zgodny z formatem IETF i mieć postać:

```
ttt, dd mmm rrrr [gg:ii:ss GMT+ppqq]
```

gdzie:

`ttt` to trzyliterowy skrót określający dzień tygodnia (Mon — poniedziałek, Tue — wtorek, Wed — środa, Thu — czwartek, Fri — piątek, Sat — sobota, Sun — niedziela),

dd to dzień tygodnia,

mmm to trzyliterowy skrót określający miesiąc (Jan — styczeń, Feb — luty, Mar — marzec, Apr — kwiecień, May — Maj, Jun — czerwiec, Jul — lipiec, Aug — sierpień, Sep — wrzesień, Oct — październik, Nov — listopad, Dec — grudzień),

rrrr to rok w formacie czterocyfrowym,

gg to godzina,

ii to minuta,

ss to sekunda,

ppqq to przesunięcie strefy czasowej, gdzie *pp* określa godziny, a *qq* minuty.

Możliwe są również skrócone postacie zapisu. Można pominać strefę czasową (zostanie wtedy przyjęty czas lokalny), a także określenie czasu (zostanie wtedy przyjęty czas 0:00:00). Istnieje także możliwość pominięcia skrótu nazwy dnia, co wymaga jednak zmiany kolejności miesiąca i numeru dnia:

mmm dd, rrrr [gg:ii:ss]

Poprawne są np. poniższe zapisy:

```
Mon, 29 Nov 2004 13:01:52 GMT+0100  
Wed, 15 Mar 2006 01:00:00  
Aug 5, 2006 05:25:24  
Jul 18, 2005
```

Wartość zwrócona przez parse może być użyta jako konstruktor innego obiektu typu Date. Konstruktor obiektów tego typu może też przyjmować ciąg znaków określający datę w opisany formacie.

Data i czas w praktyce

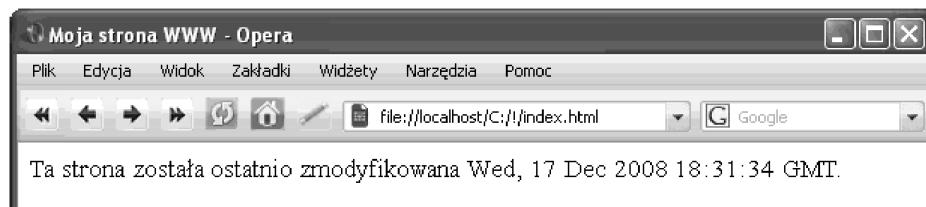
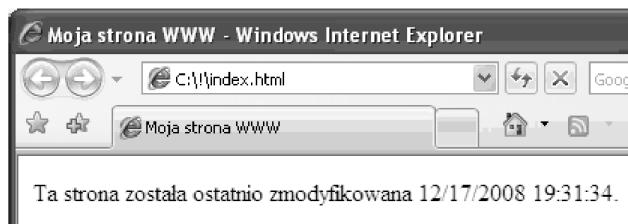
Na zakończenie 21. lekcji wykonajmy dwa przykłady korzystające w praktyce z obiektów typu Date. Na witrynach często umieszcza się informacje o datach ich ostatniej modyfikacji. Oczywiście, nie wprowadza się tej informacji ręcznie. W prostych witrynach korzystamy po prostu z właściwości `LastModified` obiektu `document` (tabela 4.4 w lekcji 12.). Zawiera ona ciąg znaków opisujący żądaną przez nas informację. Niestety, jeśli użyjemy jej bezpośrednio, np. pisząc:

```
var str = "";  
str += "Ta strona została ostatnio zmodyfikowana ";  
str += document.lastModified + ":";
```

efekty będą zależały od tego, jakiej użyliśmy przeglądarki. Różne typy przeglądarek, a także różne wersje przeglądarek z tej samej rodziny mogą bowiem prezentować dane w odmiennej postaci. Widać to na rysunkach 6.6 i 6.7. Jeśli więc chcemy, aby dane były prezentowane zawsze tak samo, w zaakceptowanym przez nas formacie, musimy napisać własny skrypt. Może on mieć postać przedstawioną na listingu 6.5.

Rysunek 6.6.

*Format właściwości
lastModified
przeglądarki
Internet Explorer*



Rysunek 6.7. Format właściwości *lastModified* przeglądarki *Opera*

Listing 6.5. Pobranie daty ostatniej aktualizacji strony

```
function ostatniaModyfikacja()
{
    var miesiące = new Array(
        "stycznia", "lutego", "marca", "kwietnia", "maja",
        "czerwca", "lipca", "sierpnia", "września",
        "października", "listopada", "grudnia"
    );
    var data = new Date(document.lastModified);
    var miesiac = miesiące[data.getMonth()];
    var dzien = data.getDate();
    if (dzien < 9){
        dzien = "0" + dzien;
    }
    var rok = data.getFullYear();
    return dzien + " " + miesiac + " " + rok + " roku.";
}

var str = "";
str += "Ta strona została ostatnio zmodyfikowana ";
str += ostatniaModyfikacja() + ".";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Za utworzenie ciągu opisującego datę ostatniej modyfikacji odpowiada funkcja `ostatniaModyfikacja`. Deklarujemy w niej tablicę `miesiące` zawierającą odmienione polskie nazwy miesięcy oraz, na podstawie właściwości `lastModified`, tworzymy nowy obiekt `Date`:

```
var data = new Date(document.lastModified);
```

Z tego obiektu za pomocą metod `getDate`, `getMonth` oraz `getFullYear` pobieramy składowe daty: dzień, miesiąc i rok. Wartość zwrócona przez `getDate`, czyli numer dnia, jest używana jako indeks tablicy `miesiące`:

```
var miesiąc = miesiące[data.getMonth()];
```

dzięki czemu miesiąc jest uzyskiwany w postaci tekstowej. Wszystkie pobrane w ten sposób składowe daty są łączone w jeden ciąg, który jest zwracany jako efekt działania funkcji:

```
return dzień + " " + miesiąc + " " + rok + " roku.:";
```

Dalsza część skryptu zawiera przykład użycia funkcji `ostatniaModyfikacja` do wyświetlenia komunikatu informującego o dacie ostatniej modyfikacji witryny.

W ostatnim przykładzie z tej lekcji pokażemy, jak wykonywać obliczenia operujące na składowych daty. Często bowiem spotykamy skrypty odliczające liczbę dni pozostały do jakiegoś wydarzenia, np. Nowego Roku, Wielkanocy czy też dowolnego innego. Konstrukcja takiego skryptu nie jest skomplikowana — wystarczy wykonać kilka prostych obliczeń i wyświetlić wynik na ekranie. Taki przykładowy skrypt został zaprezentowany na listingu 6.6.

Listing 6.6. Obliczanie liczby dni

```
function ileDni(rok, miesiąc, dzień)
{
    var data1 = new Date();
    var data2 = new Date(rok, miesiąc, dzień);
    var różnica = data2.getTime() - data1.getTime();
    return Math.floor(różnica / (1000 * 60 * 60 * 24));
}

var str = "";
str += "Pozostało jeszcze " + ileDni(2020, 1, 1);
str += " do 1 stycznia 2020 roku.:";

var dataDiv = document.getElementById("dataDiv");
dataDiv.innerHTML = str;
```

Za wykonanie obliczeń odpowiada funkcja o nazwie `ileDni`. Przyjmuje ona trzy argumenty, którymi są kolejno: rok, miesiąc i dzień, określające datę w przyszłości. Zadaniem funkcji jest obliczenie, ile dni pozostało od dnia bieżącego do tej daty. Data bieżąca przechowywana jest w zmiennej `data1`, natomiast data przyszła — w zmiennej `data2`. Działanie:

```
var różnica = data2.getTime() - data1.getTime();
```

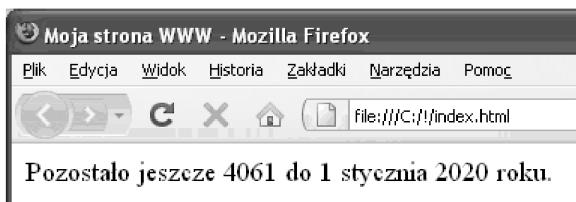
powoduje zapisanie w zmiennej `różnica` liczby milisekund, jaka upłynie między tymi datami (opis działania metody `getTime` w tabeli 6.1). Kiedy liczba milisekund jest znana, wystarczy przeliczyć je na dni, zaokraglając wynik, tak aby nie otrzymywać ułamków (jeden dzień to $1000 \times 60 \times 60 \times 24$ milisekund). Za zaokrąglenie odpowiada metoda `floor` obiektu `Math`:

```
Math.floor(różnica / (1000 * 60 * 60 * 24));
```

Obliczony w ten sposób wynik jest zwracany jako rezultat działania funkcji za pomocą instrukcji return. Funkcja `ileDni` jest wywoływana w skrypcie, a jej wynik staje się częścią wyświetlanego komunikatu. Przykładowy wynik działania takiego kodu został zaprezentowany na listingu 6.8.

Rysunek 6.8.

Efekt działania funkcji obliczającej liczbę dni



Ćwiczenia do samodzielnego wykonania

Ćwiczenie 21.1.

W oparciu o kod z listingu 6.2 napisz skrypt podający dane o dacie i czasie. Do uzyskania daty i czasu użyj osobnych funkcji.

Ćwiczenie 21.2.

Napisz kod, który doda do obiektów typu `Date` metodę pozwalającą na uzyskanie daty i czasu w Twoim własnym formacie. Możesz przystosować funkcje utworzone w ćwiczeniu 21.1.

Ćwiczenie 21.3.

Na podstawie kodu z listingu 6.3 napisz skrypt, który wyświetli nazwę bieżącego miesiąca.

Ćwiczenie 21.4.

Napisz skrypt, który będzie wyświetlał na stronie różne teksty, w zależności od dnia tygodnia, w którym użytkownik odwiedzi tę witrynę.

Ćwiczenie 21.5.

Napisz skrypt, który umożliwi użytkownikowi wprowadzenie dwóch różnych dat oraz obliczy występującą między nimi liczbę dni. Do weryfikacji poprawności formatu wprowadzanych wartości użyj wyrażeń regularnych.

Lekcja 22. Korzystanie z timerów

Lekcja 22. poświęcona jest timerom, czyli licznikom czasu. Są to mechanizmy, które pozwalają wykonywać różne operacje w zadanym czasie, czy też — dokładniej — po upływie zadanego czasu. Można więc ich używać w celu opóźnienia wykonania pewnych instrukcji bądź też do wykonywania ich cyklicznie, w określonych odstępach. Poznamy więc metody obsługujące timery, sposoby ich użycia, napiszemy przykładowe skrypty ilustrujące działanie liczników czasu, przekonamy się też, że pozwalają one na wykonywanie takich działań jak animacje.

Timery w JavaScriptie

Na korzystanie z timerów pozwalają cztery metody obiektu window (tabela 4.2 w lekcji 12.). Są to: setTimeout, clearTimeout, setInterval, clearInterval. Metody zawierające słowo set uruchamiają licznik, natomiast zawierające słowo clear przerywają jego działanie. Wywołanie setTimeout ma postać:

```
setTimeout("wyrażenie", czas)
```

co powoduje przetworzenie wyrażenia podanego jako pierwszy argument, po czasie wskazanym przez drugi argument. Czas ten należy podać w milisekundach. Przetwarzane wyrażenie może być ciągiem instrukcji, najczęściej jest to jednak nazwa funkcji, która ma zostać wykonana. Metoda setTimeout zwraca identyfikator, który może być użyty do zatrzymania odliczania czasu. W tym celu wystarczy wywołać metodę clearTimeout. Oznacza to, że jeżeli po uruchomieniu licznika, a przed upływem czasu wskazywanego przez argument *czas*, nastąpi wywołanie clearTimeout, wyrażenie nie zostanie przetworzone. Schematycznie taka konstrukcja miałaby postać:

```
var timerId = setTimeout("wyrażenie", czas)
//inne instrukcje
clearTimeout(timerId);
```

Korzystanie z clearTimeout nie jest jednak obligatoryjne.

Na listingu 6.7 jest widoczny skrypt ilustrujący działanie timera ustawianego za pomocą metody setTimeout. Współpracuje on z kodem (X)HTML o strukturze takiej samej jak przykłady z rozdziału 2.

Listing 6.7. Ilustracja działania metody setTimeout

```
function zapiszTekst()
{
    var dataDiv = document.getElementById("dataDiv");
    dataDiv.innerHTML = "Właśnie minęło 5 sekund od załadowania witryny.";
}

setTimeout("zapiszTekst()", 5000);
```

Metoda `setTimeout` jest wywoływaną z dwoma argumentami. Pierwszy to ciąg znaków `zapiszTekst()`: — jest to więc wywołanie funkcji o nazwie `zapiszTekst`. Drugi argument to wartość całkowita 5000; określa on czas, po jakim ma być przetworzone wyrażenie z pierwszego argumentu. A więc instrukcja:

```
setTimeout("zapiszTekst()", 5000);
```

oznacza po prostu: wywołaj po 5 sekundach (czyli upływie 5000 milisekund) funkcję o nazwie `zapiszTekst`. Sama funkcja `zapiszTekst` po prostu przypisuje przykładowy komunikat właściwości `innerHTML` warstwy `dataDiv`, dzięki czemu pojawia się on na ekranie.

Jak zatem będzie działał taki skrypt? Otóż, po wczytaniu zawierającej go strony do przeglądarki ujrzymy pustą witrynę. Zostanie jednak wywołana metoda `setTimeout`, co oznacza, że po 5 sekundach na warstwie `dataDiv` pojawi się tekst: „Właśnie minęło 5 sekund od załadowania witryny.”. To najlepszy dowód, że nasz timer zadziałał zgodnie z założeniami.

Obecne implementacje JavaScriptu pozwalają również, aby zamiast ciągu znaków pierwszym argumentem `setTimeout` (jak również `setInterval`) była bezpośrednio funkcja. Prawidłowa jest więc również konstrukcja o schematycznej postaci:

```
setTimeout(nazwa_funkcji, czas)
```

bądź:

```
setInterval(nazwa_funkcji, czas)
```

Wywołanie metody `setTimeout` z listingu 6.7 mogłoby zatem przyjąć również postać:

```
setTimeout(zapiszTekst, 5000);
```

Sprawdźmy teraz, jak umożliwić użytkownikowi witryny uruchamianie i zatrzymywanie timera. Można w tym celu użyć np. przycisków z odpowiednio przypisanymi procedurami obsługi zdarzeń. Przygotujmy więc kod (X)HTML takiej witryny. Przyjmie on postać widoczną na listingu 6.8.

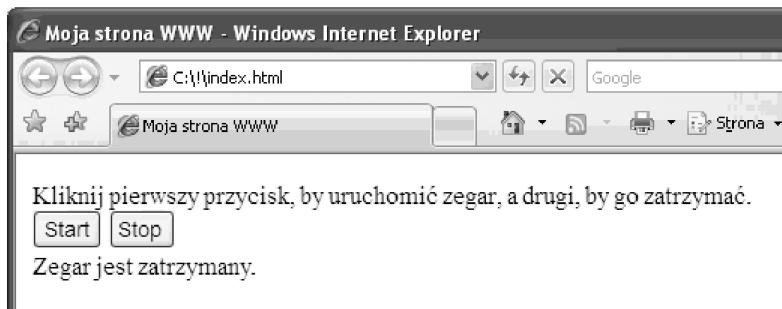
Listing 6.8. Interfejs skryptu sterującego timerem

```
<body>
  <div>
    Kliknij pierwszy przycisk, by uruchomić zegar,
    a drugi, by go zatrzymać.<br />
    <input type="button" onclick="btnStartClick();"
      id="btnStart" value="Start">
    <input type="button" onclick="btnStopClick();"
      id="btnStop" value="Stop">
  </div>
  <div id="dataDiv">
    Zegar jest zatrzymany.
  </div>
</body>
```

W kodzie znajdują się dwie warstwy. Pierwsza zawiera dwa przyciski: *Start* i *Stop*. Pierwszemu została przypisana procedura obsługi zdarzenia `onClick` w postaci funkcji `btnStartClick`; służy on do uruchamiania zegara. Procedurą obsługi zdarzenia `onClick` drugiego przycisku jest funkcja `btnStopClick`, a służy on do zatrzymywania zegara. Druga warstwa — został jej nadany identyfikator `dataDiv` — służy do wyświetlania komunikatów o stanie zegara. Po wczytaniu takiego kodu do przeglądarki zobaczymy widok zaprezentowany na rysunku 6.9. Treść skryptu JavaScript obsługującego stronę znajduje się na listingu 6.9.

Rysunek 6.9.

Przyciski umożliwiające sterowanie timerem



Listing 6.9. Kod JavaScript obsługujący timer

```
<script type="text/javascript">
    var timerId = null;
    var czas = 5000;
    function wyświetlTekst()
    {
        var dataDiv = document.getElementById("dataDiv");
        dataDiv.innerHTML = "Zegar jest zatrzymany.";
        alert("Odlicznie zostało zakończone.");
    }
    function btnStartClick()
    {
        timerId = setTimeout("wyświetlTekst()", czas);
        var dataDiv = document.getElementById("dataDiv");
        var str = "Zegar został uruchomiony na ";
        str += (czas / 1000) + " sekund.";
        dataDiv.innerHTML = str;
    }
    function btnStopClick()
    {
        if(timerId == null)
            return;
        clearTimeout(timerId);
        timerId = null;
        var dataDiv = document.getElementById("dataDiv");
        dataDiv.innerHTML = "Zegar został zatrzymany.";
    }
</script>
```

Na początku znajdują się deklaracje dwóch zmiennych: `timerId` i `czas`. Pierwsza będzie zawierała identyfikator timera uruchamianego przyciskiem *Start*. Jest to zmienna globalna, będzie więc dostępna dla wszystkich funkcji. Dzięki temu będzie możliwe zatrzymanie raz uruchomionego timera. Druga zmienna zawiera czas, na jaki ma być uruchomiony zegar (podany w milisekundach). Została umieszczona na początku skryptu dla naszej wygody, bo wtedy zmiana czasu nie będzie wymagała przeszukiwania kodu skryptu w celu odnalezienia fragmentów odwołujących się do tej danej.

Za definicjami zmiennych została umieszczona funkcja `wyświetlTekst`. Jej zadaniem jest zmiana tekstu znajdującego się na warstwie `dataDiv` na informację, że zegar jest zatrzymany, oraz wyświetlenie okna dialogowego z komunikatem o zakończeniu odliczania. Funkcja ta będzie wywoływana przez timer uruchomiony za pomocą przycisku *Start*.

Kliknięcie przycisku *Start* spowoduje wywołanie funkcji `btnStartClick`. Pierwszą i najważniejszą jej instrukcją jest:

```
timerId = setTimeout("wyświetlTekst()", czas);
```

Powoduje ona uruchomienie timera oraz przypisanie zmiennej `timerId` jego identyfikatora. Zatem po czasie określonym przez parametr `czas` zostanie wywołana funkcja `wyświetlTekst` (oczywiście, o ile zegar nie zostanie wcześniej zatrzymany). Drugim zadaniem funkcji `btnStartClick` jest wyświetlenie komunikatu o tym, że zegar został uruchomiony i na jaki czas. Czas jest podawany w sekundach, a ponieważ wartość zmiennej `czas` jest wyrażona milisekundach, do jego obliczenia jest używany wzór:

```
czas / 1000
```

Zatrzymanie zegara umożliwia funkcja `btnStopClick` wywoływana po kliknięciu przycisku *Stop*. Najpierw bada ona stan zmiennej `timerId`, czyli to, czy timer jest aktualnie uruchomiony (trwa odliczanie czasu). Jeśli `timerId` jest równe `null`, oznacza to, że timer nie został uruchomiony, a więc nie ma potrzeby go zatrzymywać — wtedy funkcja kończy działanie przez wywołanie instrukcji `return`. W przeciwnym przypadku (`timerId` różne od `null`) trwa odliczanie i należy je zatrzymać. Jest to wykonywane przez wywołanie metody `clearTimeout`:

```
clearTimeout(timerId);
```

Po jej wykonaniu zmiennej `timerId` jest przypisywana wartość `null` (co oznacza, że timer nie jest aktywny), a na warstwie `dataDiv` jest wyświetlany komunikat o zatrzymaniu zegara.

Zwróciłyśmy przy tym uwagę, że w zaprezentowanym przykładzie każde kliknięcie przycisku *Start* powoduje uruchomienie kolejnego timera. Timery są niezależne od siebie, a każdy z nich otrzymuje swój własny identyfikator. Tymczasem zapamiętujemy tylko identyfikator zwrócony przez ostatnie wywołanie metody `setTimer`. Oznacza to, że wszystkie wcześniejsze identyfikatory są tracone. Zatem kilkakrotnie kliknięcie przycisku *Start* spowoduje, że 5 sekund po każdym kliknięciu pojawi się okno dialogowe (np. 5 kliknięć to 5 okien dialogowych), a niezależnie od liczby kliknięć przycisku *Stop* można będzie zatrzymać tylko ostatnio uruchomiony zegar. Warto samodzielnie zastanowić się, w jaki sposób uniemożliwić niekontrolowaną liczbę kliknięć. Można

np. uniemożliwić uruchomienie kolejnego timera, dopóki trwa odliczanie związane z poprzednim, bądź zablokować przycisk *Start* na czas odliczania (ćwiczenie 22.1, a także przykłady z punktu „Wywołania cykliczne”).

Wywołania cykliczne (interwały)

O ile metoda `setTimeout` umożliwia jednokrotne wykonanie jakiejś instrukcji po zadanym czasie, o tyle `setInterval` pozwala na cykliczne wykonywanie takiej czynności, czyli wykonuje instrukcję co zadany czas. Zobaczmy więc, jak działa w praktyce. Odpowiedni kod został przedstawiony na listingu 6.10. Zasada jego działania jest prosta. Skrypt ma co 2 sekundy wyświetlać wartość zwiększającą się od 1 do 10. Po osiągnięciu wartości 10 odliczanie ma się rozpocząć ponownie.

Listing 6.10. Użycie metody `setInterval`

```
function zmieńTekst()
{
    var dataDiv = document.getElementById("dataDiv");
    dataDiv.innerHTML = "Wywołanie nr " + ++licznik + ".";
    if(licznik >= 10) licznik = 0;
}
var licznik = 0;
setInterval("zmieńTekst()", 2000);
```

Wykonywanie kodu rozpoczyna się od zadeklarowania globalnej zmiennej `licznik` i przypisania jej wartości 0. Następnie jest wywoływana metoda `setInterval`. Jej pierwszym argumentem jest `zmieńTekst()`, czyli instrukcja wywołująca funkcję o nazwie `zmieńTekst`. Drugi argument to wartość 2000. Ustala ona interwał, w którym ma być wykonywana instrukcja określona jako pierwszy parametr. A zatem zapis:

```
setInterval("zmieńTekst()", 2000);
```

należy po prostu rozumieć tak: „Wywołuj co 2 sekundy funkcję `zmieńTekst`”.

Zadanie funkcji `zmieńTekst` jest już bardzo proste. Ma ona:

- a)** wyświetlić wartość zmiennej `licznik`,
- b)** zwiększyć wartość tej zmiennej o 1,
- c)** wyzerować licznik, gdy osiągnie wartość 10.

Pierwsze dwa zadania są wykonywane przez instrukcję:

```
dataDiv.innerHTML = "Wywołanie nr " + ++licznik + ".";
```

a trzecie — przez instrukcję:

```
if(licznik >= 10) licznik = 0;
```

Tym samym, po uruchomieniu skryptu zobaczymy zmieniające się co 2 sekundy wartości od 1 do 10.

Sprawdźmy teraz, jak umożliwić użytkownikowi strony sterowanie timerem uruchamianym za pomocą metody setInterval. Napiszemy skrypt wyświetlający zwiększające się wartości całkowite, pozwalający na uruchamianie zegara, zatrzymywanie go, a także zerowanie licznika. Kod (X)HTML znajduje się na listingu 6.11.

Listing 6.11. Kod (X)HTML interfejsu sterującego timerem

```
<body>
<div>
    <input type="button" onclick="btnStartClick()" id="btnStart" value="Start">
    <input type="button" onclick="btnZerujClick()" id="btnZeruj" value="Zeruj">
    <input type="button" onclick="btnStopClick()" id="btnStop" value="Stop">
</div>
<div id="dataDiv">
    Zegar jest zatrzymany.
</div>
</body>
```

Struktura kodu jest taka sama jak w przykładzie z listingu 6.8. Pierwsza warstwa zawiera przyciski sterujące skryptem, a druga służy do wyświetlania danych. Tym razem przyciski są trzy. Pierwszy służy do uruchamiania timera, drugi — do zerowania licznika, a trzeci — do zatrzymywania timera. Procedurą obsługi zdarzenia onclick pierwszego przycisku jest btnStartClick, drugiego — btnZerujClick, trzeciego — btnStopClick. Funkcje te zawarte są w skrypcie przedstawionym na listingu 6.12.

Listing 6.12. Funkcje sterujące timerem

```
<script type="text/javascript">
var timerId = null;
var czas = 300;
var licznik = 0;
function wyświetlTekst(str)
{
    var dataDiv = document.getElementById("dataDiv");
    dataDiv.innerHTML = str;
}
function uaktualnijDane()
{
    licznik++;
    wyświetlTekst(licznik);
}
function btnStartClick()
{
    var btnStart = document.getElementById("btnStart");
    btnStart.disabled = true;
    timerId = setInterval("uaktualnijDane()", czas);
}
function btnStopClick()
{
```

```
clearTimeout(timerId);
timerId = null;
var btnStart = document.getElementById("btnStart");
btnStart.disabled = false;
}
function btnZerujClick()
{
    licznik = 0;
    if(timerId == null){
        wyświetlTekst(licznik);
    }
}
</script>
```

Zmienne `timerId` i `czas` pełnią tu taką samą rolę jak w przykładzie z listingu 6.9. Inna jest wartość przypisana drugiej z nich, tak by wartości zmieniały się szybciej. Trzecia zmienna — `licznik` — przechowuje wartość, która ma się pojawić na ekranie. Do dyspozycji mamy również dwie funkcje pomocnicze. Pierwsza to `wyświetlTekst`. Jej zadaniem jest wyświetlenie na warstwie `dataDiv` tekstu przekazanego w postaci argumentu `str`. Druga to `uaktualnijDane`. To ona będzie wywoływana cyklicznie po zainicjalizowaniu działania timera. A więc jej zadanie to zwiększenie wartości zmiennej `licznik` o 1 oraz wyświetlanie danych. Pierwsza z tych czynności jest wykonywana za pomocą operatora inkrementacji:

```
licznik++;
```

a druga to po prostu wywołanie funkcji `wyświetlTekst`:

```
wyświetlTekst(licznik);
```

Jeśli ktoś lubi zwięzłość zapisu, obie operacje można by przeprowadzić przy użyciu jednej instrukcji:

```
wyświetlTekst(++licznik);
```

Pozostałe funkcje to procedury obsługi zdarzeń poszczególnych przycisków. I tak `btnStartClick` jest wywoływana po kliknięciu przycisku *Start*. Oczywiście jest, że uruchamia timer za pomocą wywołania metody `setInterval`:

```
timerId = setInterval("uaktualnijDane()", czas);
```

Zatem po kliknięciu przycisku *Start* co 300 milisekund (wartość zmiennej `czas`) będzie wywoływana funkcja `uaktualnijDane`. To jednak nie wszystko, bo `btnStartClick` wyłącza również przycisk *Start*, tak by uniemożliwić uruchomienie kolejnego timera w czasie, gdy inny już pracuje. Czynność ta jest wykonywana przez przypisanie wartości `true` właściwości `disabled` przycisku `btnStart`:

```
btnStart.disabled = true;
```

Funkcja `btnStopClick` jest wywoływana po kliknięciu przycisku *Stop*. Oto zadania, które wykonuje.

1. Wyłączenie timera o identyfikatorze zapisanym w timerId.
2. Przypisanie zmiennej timerId wartości null (co jest sygnałem dla pozostałych funkcji, że timer został wyłączony).
3. Włączenie przycisku *Start* (co odbywa się przez przypisanie wartości false jego właściwości disabled).

Najprostsze zadanie ma funkcja `btnZeroj`. Ma po prostu wyzerować licznik, czyli przypisać zmiennej `licznik` wartość zero. Należy rozpatrzyć tu jedynie dwie sytuacje: albo zegar jest włączony, a więc uaktualnienie witryny nastąpi automatycznie wraz z kolejnym cyklem timera, albo też zegar jest wyłączony i wtedy trzeba go uaktualnić ręcznie. Dlatego też badany jest warunek `timerId == null`. Gdy jest on prawdziwy (zegar wyłączony), wykonywane jest uaktualnienie ekranu przez wywołanie funkcji `wyswietlTekst`. Warto zastanowić się, czy badanie tego warunku jest konieczne, czy lepiej bezwarunkowo wywoływać metodę `wyswietlTekst` (jak wtedy będzie zachowywał się skrypt przy zwiększeniu interwału), a może też całkowicie zrezygnować z jej wywołania?

Symulacja działania metody `setInterval`

Choć w cyklicznym wywoływaniu instrukcji zalecane jest używanie metody `setInterval`, spotyka się również rozwiązania polegające na symulowaniu jej działania przez specyficzne zastosowanie metody `setTimeout`. Otóż, wewnątrz funkcji, która ma być cyklicznie wywoływana, należy umieścić wywołanie `setTimeout` wskazujące na tę funkcję; oto schemat:

```
function nazwa_funkcji(argumenty)
{
    //instrukcje wewnętrza funkcji
    setTimeout("nazwa_funkcji()", czas);
}
```

Taka konstrukcja spowoduje, że funkcja `nazwa_funkcji` będzie wywoływana co tyle milisekund, ile wskazuje argument `czas`. Aby zobaczyć, jak taki kod wygląda w praktyce, zmodyfikujmy skrypt z listingu 6.10, tak by korzystał z przedstawionej metody. Przyjmie on wtedy postać zaprezentowaną listingu 6.13.

Listing 6.13. Symulacja metody `setInterval`

```
function zmieńTekst()
{
    var dataDiv = document.getElementById("dataDiv");
    dataDiv.innerHTML = "Wywołanie nr " + ++licznik + ".";
    if(licznik >= 10) licznik = 0;
    setTimeout("zmieńTekst()", 2000);
}
var licznik = 0;
zmieńTekst();
```

Główna zmiana polega na tym, że w głównej części skryptu bezpośrednio jest wywoływana metoda zmieńTekst, a dopiero w jej wnętrzu następuje wywołanie metody setTimeout uruchamiającej zegar. Ponieważ wywołanie to ma postać:

```
setTimeout("zmieńTekst()", 2000);
```

powoduje, że po dwóch sekundach znów zostanie wywołana metoda zmieńTekst, która zawiera kolejne wywołanie setTimeout itd.

Należy jednak zauważyć, że nie jest to dokładny odpowiednik skryptu z listingu 6.10. Tamten wykonywał pierwszą aktualizację witryny dopiero po 2 sekundach, obecny wykonuje ją natychmiast po uruchomieniu (wywołanie zmieńTekst()), a dopiero każde kolejne w dwusekundowych odstępach.

Zegary

W lekcji 21. poznaliśmy obiekt Date i metody pozwalające na uzyskiwanie informacji o dacie i czasie. Jeśli dodamy do tego informacje z tej lekcji dotyczące timerów, bez problemów umieścimy na stronie WWW zegar pokazujący aktualny czas. W tym celu wystarczy przecież co sekundę pobierać bieżące dane i wyświetlać je w oknie przeglądarki. W sekcji body umieścimy więc kod zaprezentowany na listingu 6.14.

Listing 6.14. Kod HTML sekcji body

```
<body onload="startTimer()">
<div id="dataDiv">---:---:</div>
</body>
```

Kod strony zawiera jedynie warstwę dataDiv, która służy do prezentacji zegara. Po częściowo został na niej umieszczony ciąg znaków ---:---:. Nie jest konieczny, ale może się przydać, gdyby skrypt się nie uruchomił (np. w przeglądarce była wyłączona obsługa JavaScriptu). Wtedy zamiast pustej strony pojawi się ten ciąg i będzie widać, w którym miejscu miał się znajdować zegar. Odliczanie czasu będzie natomiast włączane przez funkcję startTimer przypisaną jako procedura obsługi zdarzenia onload sekcji body (a zatem funkcja ta zostanie wywołana po załadowaniu witryny do przeglądarki).

Skrypt generujący zegar wyświetlany na warstwie dataDiv został zaprezentowany na listingu 6.15.

Listing 6.15. Skrypt generujący zegar

```
<script type="text/javascript">
function pobierzCzas()
{
    var data = new Date();
    var godzina = data.getHours();
    var minuta = data.getMinutes();
    var sekunda = data.getSeconds();
```

```
var str = "";
str += ((godzina < 10) ? "0" : "") + godzina;
str += ((minuta < 10) ? ":0" : ":") + minuta;
str += ((sekunda < 10) ? ":0" : ":") + sekunda;

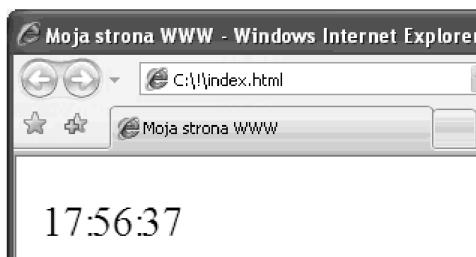
return str;
}
function uaktualnijDane()
{
    var dataDiv = document.getElementById("dataDiv");
    dataDiv.innerHTML = pobierzCzas();
}
function startTimer()
{
    uaktualnijDane();
    setInterval(uaktualnijDane, 1000);
}
</script>
```

Funkcja pobierzCzas ma za zadanie pobrać aktualny czas i zwrócić go w postaci ciągu znaków w formacie *GG:MM:SS*. Najpierw tworzy nowy obiekt typu Date zawierający bieżące dane i zapisuje go w zmiennej data. Następnie używa metod getHours, getMinutes i getSeconds do uzyskania liczby godzin, minut i sekund. Dane te zapisuje w zmiennych godzina, minuta i sekunda. Ponieważ wszystkie dane mają mieć format dwucyfrowy, gdy ktoś z nich ma wartość mniejszą od 9, na początku dodawany jest znak 0. Używany jest operator warunkowy na takiej samej zasadzie, jak miało to miejsce w przykładzie z listingu 6.2. Wszystkie dane są łączone w zmiennej str, której wartość jest ostatecznie zwracana jako wynik działania funkcji.

Funkcja uaktualnijDane wykonuje tylko jedno zadanie: ma wyświetlić na warstwie dataDiv aktualny czas. Pobiera więc odwołanie do warstwy, zapisuje je w zmiennej dataDiv oraz przypisuje wartość zwróconą przez wywołanie funkcji pobierzCzas właściwości innerHTML.

Funkcja startTimer jest wywoływaną po załadowaniu strony do przeglądarki, a jej zadaniem jest zapoczątkowanie procesu wyświetlania czasu. Najpierw wywołuje funkcję uaktualnijDane, za jej pomocą raz po załadowaniu strony na ekranie pojawi się aktualny czas. Następnie wywołuje metodę setInterval, która sprawia, że kolejne wywołania wymienionej funkcji będą następowaly co sekundę. Zatem w oknie przeglądarki zacznie działać zegar informujący o bieżącym czasie. Będzie miał postać przedstawioną na rysunku 6.10.

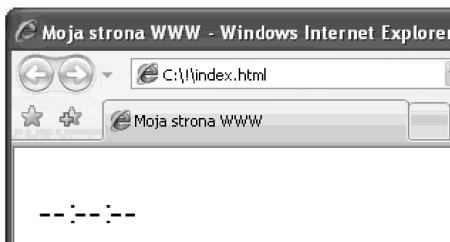
Rysunek 6.10.
Zegar generowany
przez skrypt



Gdybyśmy pominęli pierwsze, bezpośrednie wywołanie funkcji aktualnijDane, zegar zacząłby działać dopiero po sekundzie od załadowania strony. Wtedy w pierwszej chwili w oknie przeglądarki pojawiłaby się pierwotna treść warstwy dataDiv, czyli ciąg `--:--:--`, co zostało zaprezentowane na rysunku 6.11. Dopiero po chwili na ekranie ukazałby się właściwy zegar odmierzający czas.

Rysunek 6.11.

Oczekiwanie na wyświetlenie aktualnego czasu



Animacje

Skoro potrafimy uruchamiać wskazany kod w określonych odstępach czasu, możemy też wykorzystać tę umiejętność do tworzenia różnych animacji. Odpowiednio manipulując danymi, możemy osiągnąć efekt ruchu fragmentów witryny, zmieniających się kolorów, pływających napisów itp. Za animację można już uznać prezentowany w poprzednim punkcie lekcji zegar. Bardziej zaawansowane efekty nie muszą być wcale skomplikowane. Zastanówmy się np., jak można osiągnąć płynną zmianę koloru? Standardowo pojedynczy kolor jest opisywany przez składowe RGB (czerwoną, zieloną i niebieską). Obowiązuje przy tym model, w którym każda składowa zapisywana jest na 8 bitach, czyli może przyjmować wartości od 0 do 255. Przykładowo składowe `RGB(255, 0, 0)` to kolor czerwony, `RGB(0, 255, 0)` — zielony, a `RGB(128, 128, 128)` — szary. Skoro tak, to w JavaScriptie w prosty sposób możemy zmieniać kolor danego elementu, pisząc:

```
element.style.color = "RGB(r, g, b)";
```

A także jego tło:

```
element.style.backgroundColor = "RGB(r, g, b)";
```

gdzie `element` to obiekt odzwierciedlający dany element witryny, a `r, g i b` to poszczególne składowe.

Jeżeli teraz wybrane składowe koloru będziemy zmieniać w niewielkich odstępach czasu, osiągniemy efekt płynnej zmiany koloru. Sprawdźmy to w praktyce. Będziemy modyfikować kolor tła przykładowej warstwy. Umieszcimy ją w sekcji body strony (X)HTML, co zostało zaprezentowane na listingu 6.16.

Listing 6.16. Definicja warstwy płynnie zmieniającej kolor

```
<body onload="startAnim();">
  <div id="dataDiv" style="width:100px;height:100px;">
  </div>
</body>
```

Warstwie został nadany identyfikator dataDiv oraz styl określający jej wysokość i szerokość na 100 pikseli. Zdarzeniu onload sekcji body została przypisana procedura obsługi w postaci funkcji startAnim. Będzie ona odpowiedzialna za uruchomienie animacji. Skrypt obsługujący animację został przedstawiony na listingu 6.17.

Listing 6.17. Skrypt wykonujacy animacje kolorów

```
<script type="text/javascript">
var czas = 100;
var krok = 5;
var rgb = 0;
var dataDiv = null;

function zmieńKolor()
{
    var color = "RGB(";
    color += rgb + ", 0, " + (255-rgb) + ")";
    dataDiv.style.backgroundColor = color;
    rgb += krok;
    if(rgb > 255){ rgb = 255; krok = -krok;};
    if(rgb < 0){ rgb = 0; krok = -krok;};
}
function startAnim()
{
    dataDiv = document.getElementById("dataDiv");
    setInterval(zmieńKolor, czas);
}
</script>
```

Na początku kodu znajdują się definicje zmiennych globalnych:

1. czas — określa czas pomiędzy kolejnymi krokami animacji,
2. krok — określa, o ile będą się zmieniać wybrane składowe RGB w każdym kroku animacji,
3. rgb — określa aktualną wartość danej składowej RGB,
4. dataDiv — przechowuje odwołanie do warstwy dataDiv.

Funkcja startAnim jest wywoływana po załadowaniu strony i ma za zadanie rozpoczęć animację. Najpierw jednak pobiera odwołanie do warstwy dataDiv i zapisuje je w zmiennej dataDiv, co sprawia, że operacja ta nie będzie musiała być wykonywana w każdym wywołaniu funkcji obsługującej zmiany koloru. Dopiero w drugim kroku wywoływana jest metoda setInterval, która powoduje, że w odstępach czasu określonych przez zmienną czas będzie cyklicznie wywoływana funkcja zmieńKolor. Został tu użyty drugi ze sposobów wywołania metody setInterval, w którym jako pierwszy argument występuje funkcja (obiekt funkcji), a nie, jak do tej pory, ciąg znaków:

```
setInterval(zmieńKolor, czas);
```

Funkcja zmieńKolor przy każdym swoim wywołaniu ma zmienić kolor tła warstwy dataDiv, tak by w efekcie powstała płynna animacja. Konstruuje ona ciąg znaków, który zostanie przypisany właściwości backgroundColor obiektu style warstwy dataDiv (przykłady z lekcji 16.). Jest on zapisywany w zmiennej color i ma format:

RGB(*r*, *g*, *b*)

Odpowiedzialne są za to instrukcje:

```
var color = "RGB(";  
color += rgb + ", 0, " + (255-rgb) + ")";
```

Oznacza to, że składowa *G* pozostaje stała, równa 0. Składowa *R* przyjmuje wartość zapisaną w zmiennej *rgb*, a składowa *B* — wartość wynikającą ze wzoru 255 - *rgb*. Gdy składowa *R* zwiększa się, składowa *B* maleje. I odwrotnie, gdy składowa *R* się zmniejsza, składowa *B* zwiększa się. Przypisanie tak określonego koloru właściwości backgroundColor odbywa się w sposób standardowy:

```
dataDiv.style.backgroundColor = color;
```

Następnie jest uaktualniana wartość zmiennej *rgb* poprzez dodanie do niej wartości zapisanej w krok. Oznacza to, że im większa będzie wartość krok, tym szybciej będą następowały zmiany (jednocześnie przejście między kolejnymi kolorami będą mniej płynne, bowiem większy będzie odstęp między poszczególnymi odcieniami).

Kolor ma się zmieniać od wartości RBG(0, 0, 255) do RBG(255, 0, 0), a następnie z powrotem, od RBG(255, 0, 0) do RBG(0, 0, 255). Osiągamy to, manipulując znakiem zmiennej krok. Gdy jest dodatnia, wartość *rgb* zwiększa się, gdy jest ujemna, wartość *rgb* zmniejsza się. Trzeba jedynie wykryć moment, kiedy powinien zostać zmieniony znak. To nie jest skomplikowane. Wartościami granicznymi dowolnej składowej RGB są 0 i 255. A zatem jeżeli wartość zapisana w *rgb* przekroczy 255:

```
if(rgb > 255){
```

należy ją zamienić na 255:

```
rgb = 255;
```

oraz zmienić znak wartości zapisanej w krok:

```
krok = -krok;
```

Z analogiczną sytuacją mamy do czynienia, gdy wartość zapisana w *rgb* będzie mniejsza niż 0. Należy wtedy przypisać jej wartość 0 i zmienić znak wartości zapisanej w *krok*.

Funkcja startAnim wykonuje tylko dwa zadania: pobiera odwołanie do warstwy dataDiv i zapisuje je w zmiennej o takiej samej nazwie oraz uruchamia animację, wywołując metodę setInterval. Ponieważ jest ona procedurą obsługi zdarzenia onload sekcji body, animacja uruchomi się tuż po załadowaniu strony do przeglądarki.

Zastanówmy się teraz, jak wprawić warstwę w ruch. Nie wydaje się to skomplikowane. Styl przypisany warstwie określa jej położenie na witrynie. Jeżeli ustalimy pozycjonowanie bezwzględne (position:absolute), będziemy mogli swobodnie przesuwać

warstwę po całym obszarze strony, manipulując właściwościami `left` i `top` obiektu `style`. Napiszmy zatem skrypt, który spowoduje, że przykładowa warstwa będzie się poruszała w wyznaczonym obszarze strony. Kod X(HTML) będzie miał postać zaprezentowaną na listingu 6.18.

Listing 6.18. Warstwa, która będzie się poruszała wewnątrz witryny

```
<body onload="startAnim();">
<div id="dataDiv"
    style="width:80px;height:20px;left:0px;top:0px;background-color:yellow;
           border:1px solid silver;position:absolute;">
</div>
</body>
```

Zdarzeniu `onload` sekcji `body` została przypisana procedura obsługi w postaci funkcji `startAnim`. Funkcja ta będzie odpowiedzialna za uruchomienie animacji. Sama warstwa została zdefiniowana w sposób klasyczny, za pomocą znacznika `<div>` z atrybutem `id` ustawionym na `dataDiv`. Został jej również przypisany styl określający rozmiary (80×20 pikseli), położenie (współrzędne lewego górnego rogu: $x = 0$, $y = 0$), kolor tła (żółty), obramowanie (czarne, pełne, o szerokości 1 piksela), pozycjonowanie (bez względne).

Aby poruszać taką zdefiniowaną warstwą, trzeba napisać skrypt zmieniający współrzędne x i y lewego górnego rogu (czyli właściwości `left` i `top`). Będzie on działał tak, by warstwa poruszała się samoistnie po ograniczonym obszarze, odbijając się od jego krańców. Skrypt został przedstawiony na listingu 6.19.

Listing 6.19. Skrypt poruszający warstwę

```
<script type="text/javascript">
var dataDiv = null;
var stepX = 3;
var stepY = 2;
var szybkość = 50;

function przesuń()
{
    var x = parseInt(dataDiv.style.left);
    var y = parseInt(dataDiv.style.top);
    x += stepX;
    y += stepY;
    if(x > 200){x = 200; stepX = -stepX;}
    if(y > 200){y = 200; stepY = -stepY;}
    if(x < 0){x = 0; stepX = -stepX;}
    if(y < 0){y = 0; stepY = -stepY;}
    dataDiv.style.left = x + "px";
    dataDiv.style.top = y + "px";
}
function startAnim()
{
    dataDiv = document.getElementById("dataDiv");
    setInterval(przesuń, szybkość);
}
```

Na początku znajdują się definicje czterech zmiennych globalnych:

1. `dataDiv` to odwołanie do przesuwanej warstwy o identyfikatorze `dataDiv`;
2. `stepX` to liczba pikseli, o którą warstwa zostanie przesunięta w poziomie, w każdym kroku;
3. `stepY` to liczba pikseli, o którą warstwa zostanie przesunięta w pionie, w każdym kroku;
4. `czas` to liczba milisekund pomiędzy kolejnymi krokami animacji.

Animacja jest uruchamiana przez funkcję `startAnim` wywoływaną automatycznie po załadowaniu witryny do przeglądarki. Funkcja inicjalizuje zmienną `dataDiv`, przypisując jej odwołanie do warstwy o takim samym identyfikatorze, oraz uruchamia `timer` powodujący wywoływanie funkcji `przesuń` w odstępach określonych przez wartość zmiennej `czas`.

Zadaniem funkcji `przesuń` jest przemieszczanie przy każdym wywołaniu warstwy `dataDiv` o liczbę pikseli zapisaną w `stepX` i `stepY`. Najpierw odczytuje więc aktualną wartość zapisaną we właściwościach `left` (współrzędna `x`) oraz `top` (współrzędna `y`) obiektu `style`. Ponieważ są to ciągi znaków określające zarówno wartość, jak i jednostkę miary (w naszym przypadku piksele — `px`), dane są przetwarzane za pomocą metody `parseInt` i zapisywane w pomocniczych zmiennych lokalnych `x` i `y`:

```
var x = parseInt(dataDiv.style.left);
var y = parseInt(dataDiv.style.top);
```

Następnie wartości, które znalazły się w `x` i `y`, są zwiększane o wartości zapisane w `stepX` (przesunięcie w poziomie) i `stepY` (przesunięcie w pionie):

```
x += stepX;
y += stepY;
```

Aby przesunąć warstwę, wystarczyłoby tak uzyskane wartości ponownie zapisać we właściwościach `top` i `left`. Trzeba jednak wziąć pod uwagę, że ruch ma się odbywać w ograniczonym obszarze strony. W przykładzie założono, że współrzędna `x` nie może być mniejsza niż 0 (czyli warstwa nie może się znaleźć poza lewą krawędzią przeglądarki) oraz większa niż 200. Podobnie, współrzędna `y` nie może być mniejsza od 0 (czyli warstwa nie może się znaleźć poza górną krawędzią przeglądarki) oraz większa niż 200. Oczywiście, można te wartości zmieniać, zmniejszając lub zwiększając wyznaczony obszar.

Skoro jednak opisane ograniczenie ma wstępuwać, niezbędne jest zastosowanie serii instrukcji warunkowych badających stan współrzędnych. Gdy zostanie wykryte, że dana współrzędna wykracza poza wyznaczony obszar, trzeba ją odpowiednio ograniczyć oraz zmienić znak wartości zapisanej w `stepX` bądź `stepY`. Jeśli np. `x` jest większe od 200, znaczy to, że `stepX` było dodatnie, a warstwa poruszała się w poziomie w prawo. Ponieważ wartość `x` nie może być większa niż 200, zmiennej `x` przypisywana jest wartość 200 oraz na przeciwny zmieniany jest znak `stepX`:

```
if(x > 200){x = 200; stepX = -stepX;}
```

Tym samym, stepX będzie ujemne, a warstwa zacznie się poruszać w poziomie w lewo.

Kiedy wszystkie warunki zostaną zbadane, mamy pewność, że zarówno w x, jak i w y znajdują się właściwe wartości, mogą więc być przypisane właściwościom left i top. Trzeba jedynie pamiętać, że należy przypisać nie wartość całkowitą, ale ciąg określający położenie wraz z jednostką miary. Ponieważ w naszym przypadku są to piksele oznaczane jako px, przypisanie odbywa się w następujący sposób:

```
dataDiv.style.left = x + "px";
dataDiv.style.top = y + "px";
```

To wszystko. Wczytanie strony do przeglądarki spowoduje uruchomienie skryptu, a tym samym warstwa dataDiv zacznie „pływąć” w wyznaczonym obszarze, odbijając się od jego krańców.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 22.1.

Popraw kod z listingu 6.9, tak by nie występował problem wielokrotnego kliknięcia przycisku *Start*. Dodatkowo przycisk *Stop* powinien być aktywny tylko w trakcie odliczania czasu (aktywności timera).

Ćwiczenie 22.2.

Zmodyfikuj kod z listingu 6.9, tak by kliknięcie przycisku *Start* w trakcie działania timera zatrzymywało go oraz rozpoczynało kolejny cykl odliczania.

Ćwiczenie 22.3.

Napisz skrypt zachowujący się podobnie jak przykład z listingu 6.10, który jednak po osiągnięciu wartości 10 zaczyna odliczanie w dół do 1, po osiągnięciu wartości jeden odlicza ponownie do 10 itd.

Ćwiczenie 22.4.

Na podstawie przykładu z listingów 6.16 i 6.17 napisz skrypt, w którym warstwa płynnie zmienia kolory, ale użytkownik ma możliwość uruchamiania i zatrzymywania animacji.

Ćwiczenie 22.5.

Zmodyfikuj kod przykładu z listingów 6.18 i 6.19, tak by animacja warstwy była uruchamiana i zatrzymywana przez jej kliknięcie.

Skorowidz

!, 42
!=, 41
!==, 41
&, 44
&&, 42
/* */, 29
//, 29
?:, 62
^, 44
|, 44
||, 42
~, 44
++, 71
<!-- -->, 29
<<, 44, 46
, 27, 256
<blink>, 256
<body>, 13

, 27
<form>, 269, 274
<head>, 13
<i>, 27, 256
<input>, 220, 256
 accessKey, 220
 alt, 220
 blur(), 222
 checked, 220
 className, 220
 click(), 222
 cols, 221
 defaultChecked, 221
 defaultValue, 221
 dir, 221
 disabled, 221
 focus(), 222
 form, 221
 id, 221
 lang, 221
 length, 221
 maxLength, 221
 multiple, 221
 name, 221
 readOnly, 221
 remove(), 222
 rows, 221
 selectedIndex, 221
 size, 221
 tabIndex, 221
 title, 221
 type, 221
 type="button", 222
 type="checkbox", 224
 type="password", 234
 type="radio", 227
 type="text", 229
 value, 221
<meta>, 10
<noscript>, 9, 15
<option>, 235
<script>, 9, 11, 20, 29
 language, 12
<select>, 235
<strike>, 256
<style>, 202
<sup>, 27, 102
<textarea>, 232, 266
=, 46
==, 41
====, 41
>>, 44, 46
>>>, 44

A

abs(), 103
 acceptCharset, 270
 accessKey, 220
 acos(), 103
 action, 269, 270
 ActionScript, 17
 adres URL, 96
 adres URL aktualnie załadowanego dokumentu, 185
 AJAX, 7
 akapity tekstowe udające odnośniki, 212
 alert(), 78, 172, 175
 alinkColor, 182
 all, 182
 alt, 220
 anchor(), 257
 anchors, 182
 AND, 42, 44
 animacja, 352, 356
 płynna zmiana koloru, 352
 appCodeName, 188
 appendChild(), 203, 219
 applets, 182
 application/ecmascript, 12
 application/javascript, 12
 appMinorVersion, 188
 appName, 188
 appVersion, 188
 arguments, 125, 127
 argumenty, 79, 119, 125
 funkcje, 122
 liczba, 126
 niezdefiniowane, 87
 sposób przekazywania, 85
 Array, 140, 150
 asin(), 103
 assign(), 186
 atan(), 103
 atan2(), 103
 atrybuty stylu CSS, 284
 automatyczny odczyt właściwości, 115
 automatyczny zapis właściwości, 115

B

back(), 175, 185
 background-color, 242, 244, 245, 250
 backgroundColor, 354
 badanie liczby argumentów, 126
 bezpośrednia manipulacja węzłami dokumentu, 196

bezpośrednie przypisywanie właściwości, 111
 bgColor, 182
 białe znaki, 30
 bieżąca data, 327
 big(), 257
 bitowa różnica symetryczna, 45
 bity, 43
 blink(), 257
 blok obsługi wyjątku, 169
 blokada wyskakujących okien, 181
 blur(), 175
 błędы, 161
 sygnalizacja, 163
 body, 182
 bold(), 257
 boolean, 26
 border, 250
 break, 60, 61, 72

C

callee, 127
 captureEvents(), 175
 case, 60
 catch, 159, 165
 ceil(), 103, 105
 characterSet, 182
 charAt(), 259, 260
 charCodeAt(), 259, 260
 charset, 12
 checkbox, 224
 checked, 220
 childNodes, 197
 chrome, 180
 ciągi znaków, 23, 24, 253
 długość, 253
 dzielenie względem znaków separatora, 264
 formatowanie, 256
 length, 253
 łączenie, 50
 metody formatujące, 256
 metody przetwarzające, 259
 operacje, 50
 pobieranie fragmentu ciągu, 263
 przeszukiwanie, 262
 przetwarzanie, 259
 string, 253
 wyrażenia regularne, 262
 zamiana podciągów, 262
 class, 249
 className, 220, 249, 270
 clearInterval(), 175, 342

clearTimeout(), 175, 342, 345
close(), 175
closed, 173
compatMode, 182
compile(), 287
concat(), 149, 151, 259
confirm(), 175, 177
constructor, 134
continue, 74
cookie, 182, 315
cookieEnabled, 188
cookies, 311
 bezpieczne połączenia, 312
 czas ważności, 324
 czas życia, 312, 313
 domena, 312, 314
 expires, 313
 HTTPS, 312
 JavaScript, 315
 liczba odwiedzin, 323
 max-age, 313
 nagłówek HTTP, 312
 nazwa, 312
 odczytywanie, 316, 320
 podglądarka, 315
 ścieżka dostępu, 312
 wartość, 312
 zapisywanie, 315, 317
cos(), 103
cpuClass, 188
createElement(), 198, 199, 218
createTextNode(), 198, 199, 219
CriScript, 17
CSS, 201, 240
cssText, 281
cykliczne wykonywanie czynności, 346
czas, 327, 338
 pobieranie, 331
 pora dnia, 336
 UTC, 328

D

data, 327, 338
 bieżąca, 327
Date, 327
 dzień tygodnia, 334
formatowanie, 333, 337
obliczenia, 340
ostatnia aktualizacja strony, 339
pobieranie, 331

Date, 327
 getDate(), 328
 getDay(), 328
 getFullYear(), 328
 getHours(), 328
 getMilliseconds(), 328
 getMinutes(), 328
 getMonth(), 329
 getSeconds(), 329
 getTime(), 329
 getTimezoneOffset(), 329
 getYear(), 329
 parse(), 329, 337
 setDate(), 329
 setFullYear(), 329
 setHours(), 329
 setMilliseconds(), 329
 setMinutes(), 330
 setMonth(), 330
 setSeconds(), 330
 setTime(), 330
 setYear(), 330
 toDateString(), 330
 toGMTString(), 330
 toLocaleDateString(), 331
 toLocaleString(), 331
 toLocaleTimeString(), 331
 toString(), 331
 toTimeString(), 331
 toUTCString(), 331
decodeURI(), 92, 97
decodeURIComponent(), 92, 97
default, 60
defaultStatus, 173
defer, 12
deklaracja typu dokumentu, 10
deklaracja zmiennej, 22, 27
dekrementacja, 38
dependent, 180
dialog, 180
dir, 270
directories, 180
disabled, 348
disableExternalCapture(), 175
długość tablicy, 144
długość tekstu, 253
do...while, 67
docType, 182
DOCTYPE, 10
document, 173, 181
 cookie, 315, 316
createElement(), 198

document, 173, 181
 createTextNode(), 198
 forms, 272
 getElementById(), 22, 193
 metody, 184
 właściwości, 182
 write(), 32
 Document Object Model, 171, 191
 dokumenty (X)HTML, 191
 węzły, 192
 węzły tekstowe, 192
 DOM, 171, 190, 191
 DOM Inspector, 191
 domain, 182
 dostęp do argumentów funkcji, 125
 dostęp do atrybutów CSS, 240
 dostęp do elementów strony WWW, 193
 dostęp do elementów witryny, 190
 dostęp do obiektów formularzy, 269
 dostęp do właściwości obiektów, 113
 drzewo DOM, 191, 194
 węzły, 192
 dynamiczna zmiana stylów CSS, 248
 dynamiczne elementy dokumentu, 218
 dynamiczne przypisywanie procedur obsługi
 zdarzeń, 215
 dzień tygodnia, 334

E

E, 101
 ECMA, 17
 ECMA v3, 32
 ECMAScript, 17
 ECMAScript v3, 31
 edycja atrybutów stylu CSS, 245
 EJScript, 17
 elements, 270
 elementy witryny, 220
 embeds, 182
 enableExternalCapture(), 175
 encodeURI(), 92, 96
 encodeURIComponent(), 92
 encoding, 270
 enctype, 269, 270
 Enter, 31
 Error, 158, 160, 169
 escape(), 92, 97
 eval(), 92, 97, 109
 EvalError, 169, 170
 event, 173, 204
 event handler, 204, 205
 exception handler, 169
 exceptions, 157

exec(), 287, 299, 310
 exp(), 103
 expires, 313

F

false, 26
 fałsz, 26
 fgColor, 182
 finally, 165
 find(), 175
 fixed(), 257
 floor(), 103, 105
 focus(), 176
 fontcolor(), 257
 fontsize(), 257
 for, 64
 warianty pętli, 75
 for...in, 69, 148
 form, 270, 271
 formatowanie
 ciągi znaków, 27, 256
 daty, 337
 forms, 182, 269, 272
 formularze, 269
 dostęp do obiektów, 269
 sprawdzanie poprawności danych, 273
 właściwości, 270
 forward(), 176, 185
 frames, 173
 fromCharCode(), 259, 261
 function, 78, 121
 funkcje, 77, 78, 121
 argumenty, 79, 125
 globalne, 92
 pomijanie argumentów, 86
 przekazywanie argumentów, 85
 przekazywanie danych, 79
 przypisanie zmiennej, 122
 return, 81
 tworzenie, 78
 wewnętrzne, 89
 właściwości, 125
 wywołanie, 79
 zasięg zmiennych, 82
 zwracanie wartości, 81

G

getAttribute(), 240, 242, 243, 244
 getDate(), 328
 getDay(), 328
 getElementById(), 184, 193, 197, 215, 240

getElementsByName(), 184
getElementsByTagName(), 184
getFullYear(), 328
getHours(), 328
getMilliseconds(), 328
getMinutes(), 328
getMonth(), 329
getSeconds(), 329
getTime(), 329
getTimezoneOffset(), 329
getYear(), 329
global, 287
global object, 127
go(), 185

H

handleEvent(), 176
hash, 185
hasła, 233
height, 179, 182, 250
historia odwiedzin stron, 184
history, 173, 184
 back(), 185
 current, 184
 forward(), 185
 go(), 185
 length, 184
 next, 184
 previous, 184
 właściwości, 184
home(), 176
host, 185
hostname, 185
href, 185
HTML, 11
HTML 4.01, 9
HTTP_USER_AGENT, 188
HTTPS, 312

I

id, 270
identifiers, 30
identyfikatory, 30
if, 51
 else, 54
 else if, 55
ignoreCase, 287
iloczyn bitowy, 43
iloczyn logiczny, 42
images, 182
indeks tablicy, 144

indeksowanie tablicy, 141
indexOf(), 259, 263, 267, 318, 321
Infinity, 100, 104, 162
informacje
 data i czas, 331, 334
 dokument, 183
 formularz, 272
 przeglądarka, 187
inkrementacja, 38
innerHeight, 173, 180
innerHTML, 22, 194, 195, 196, 212, 255, 279,
 303, 332
innerWidth, 173, 180
input, 287
Inspektor DOM, 191
instrukcje, 21
 if, 51
 if...else, 54
 if...else if, 55
 switch, 60
 warunkowe, 51
 zagnieżdzanie instrukcji warunkowych, 56
interwały, 346
isFinite(), 96, 161
isNaN(), 92, 161
italics(), 257
iteracja, 65

J

Java, 17
JavaScript, 7, 17
JavaScript Object Notation, 107
język JavaScript, 7
join(), 150
JScript, 17
JSON, 107

K

klasy znakowe, 291
kliknięcie, 211
kod HTML, 20
kod pocztowy, 308
kod XHTML, 20
kodowanie ciągów znaków, 96
kodowanie MIME, 269
kolory, 354
komentarze, 28
 blokowe, 29
 liniowe, 29
 wierszowe, 29

komunikaty o błędzie, 163
 Konsola błędów, 158
 konstruktory, 129
 metody, 133
 kontynuacja pętli, 74
 konwersja ciągu znaków na liczbę całkowitą, 93
 kotwice, 298

L

lang, 270
 language, 12, 188
 lastIndex, 287
 lastIndexOf(), 259, 262
 lastMatch, 287
 lastModified, 182, 339
 lastParen, 287
 left, 179
 leftContext, 287
 length, 127, 144, 173, 253, 270
 liczba dni, 340
 liczba odwiedzin, 323
 liczby, 23, 24
 link(), 257
 linkColor, 182
 links, 182
 lista wyboru, 235
 literal constant, 24
 literały, 24
 LN10, 101
 LN2, 101
 location, 173, 180, 182, 185
 assign(), 186
 hash, 185
 host, 185
 hostname, 185
 href, 185
 metody, 186
 pathname, 185
 port, 185
 protocol, 185
 reload(), 186
 replace(), 186
 search, 185
 toString(), 186
 właściwości, 185
 locationbar, 173
 log(), 103
 LOG10E, 101
 LOG2E, 101

ładowanie strony, 207
 łańcuchy, 24
 łączenie
 ciagi znaków, 50
 tablice, 149

Ł

manipulowanie elementami strony WWW, 193
 manipulowanie stylami CSS, 241
 manipulowanie węzłami dokumentu, 196
 match(), 259, 262, 305, 306
 Math, 100
 ceil(), 105
 E, 100
 floor(), 105
 PI, 100
 pow(), 105
 round(), 105
 sqrt(), 58, 105
 max(), 103
 max-age, 313
 menubar, 173, 180
 metaznaki, 290
 method, 269, 270
 metody, 116
 argumenty, 119
 wywołanie, 118
 MIME, 269
 mimetypes, 188
 min(), 103
 minimizable, 180
 modal, 180
 model DOM, 171, 191
 modulo, 38
 modyfikacja stylów CSS, 241
 Internet Explorer, 244
 moveBy(), 176
 moveTo(), 176
 multiline, 287
 multiple, 236
 name, 173, 270
 NaN, 100, 105
 napisy, 24
 navigator, 187
 właściwości, 188

M**N**

nazwy
argumenty, 86
zmienne, 23, 86
zmienne iteracyjne, 69
negacja bitowa, 45
negacja logiczna, 42
new, 129, 134
niezdefiniowane argumenty, 87
nodeValue, 197
NOT, 42, 44, 45
notacja JSON, 107
null, 26

O

obiekt globalny, 127
obiekty, 26, 91, 106, 107
 Array, 140
 bezpośrednie przypisywanie właściwości, 111
 Date, 327
 document, 181
 dostęp do właściwości, 113
 Error, 158, 169
 form, 270, 271
 forms, 269
 funkcje, 116
 history, 184
 JSON, 107
 konstruktory, 129
 location, 185
 metody, 116
 navigator, 187
 odczyt danych, 114
 option, 238
 przesłanianie właściwości, 136
 RegExp, 286
 składowe, 107, 108
 string, 253
 style, 244
 this, 116, 127
 window, 172
 właściwości, 109
 wywołanie metody, 118
 zagnieżdzanie, 128
 zapis danych, 114
obiekty główne przeglądarki, 172
Object, 129
obliczanie liczby dni, 340
obramowanie, 250
obsługa błędów, 157, 161
obsługa daty i czasu, 327
obsługa zdarzeń, 204
odczytywanie cookies, 316, 320

odnośniki, 212
odwracanie kolejności elementów tablicy, 151
okna, 179
okno dialogowe, 9
onabort, 206
onblur, 206
onchange, 206, 236
onclick, 205, 206, 211, 212, 223, 248
ondblclick, 206
onerror, 206
onfocus, 206
onkeydown, 206, 255
onkeypress, 206, 255
onkeyup, 206, 255
online, 188
onload, 207, 209, 350
onmousedown, 207
onmousemove, 207
onmouseout, 207, 213, 215, 248
onmouseover, 207, 213, 215, 248
onmouseup, 207
onreset, 207
onresize, 207
onselect, 207
onsubmit, 207, 276, 277
onunload, 207, 210
open(), 176, 179
opener, 173
operacje, 33
operacje na ciągach znaków, 50, 253
operatory, 33, 49
 arytmetyczne, 36
 bitowe, 43, 44
 dekrementacja, 38
 inkrementacja, 38
 logiczne, 41
 new, 129
 porównywanie, 41
 priorytety, 49
 przypisanie, 46
 relacyjne, 41
 typeof, 48
 warunkowy operator, 62
option, 238
OR, 42, 44
oscpu, 188
outerHeight, 173, 179
outerWidth, 173, 179

P

pageXOffset, 173
pageYOffset, 173
parent, 173

parse(), 328, 329, 337
parseFloat(), 92
parseInt(), 91, 94, 127
pathname, 185
personalbar, 173, 180
pętle, 64, 114
 break, 72
 continue, 74
 do...while, 67
 for, 64
 for...in, 69, 148
 iteracja, 65
 kontynuacja wykonania, 74
 nazwy zmiennych iteracyjnych, 69
 przebieg, 65
 przerywanie działania, 72
 while, 66
 zagieźdzanie pętli, 70
 zmienna iteracyjna, 65
PI, 101
platform, 188
plugins, 182, 188
płynna zmieniana koloru, 352
pobieranie
 czas, 331
 data, 331
podglądarki cookies, 315
pole tekstowe, 229
 password, 233
 przetwarzanie tekstu, 258
pole wyboru
 checkbox, 224
 radio, 227
polskie litery, 31
pomijanie argumentów, 86
pop(), 150, 151
poprawność hasła, 235
pora dnia, 336
porównywanie, 41
port, 185
potwierdzenie operacji, 177
pow(), 103
prawda, 26
predefiniowane obiekty wyjątków, 169
print(), 176
priorytety operatorów, 49
procedury obsługi zdarzeń, 204, 209
product, 188
productSub, 188
programowe usunięcie elementów dokumentu, 200
prompt(), 176, 177

propagacja wyjątków, 169
protocol, 185
prototypy, 132
przebieg pętli, 65
przechowywanie danych, 22
przechwytywanie wyjątków, 159
przeglądarki, 187
 rozpoznanie typu, 189
przekazywanie argumentów, 85
 przez referencję, 85
 przez wartość, 85
przełączanie wezłów DOM, 203
przerywanie pętli, 72
przesłanianie właściwości, 136
przesunięcie bitowe, 44, 46
przetwarzanie
 ciagi znaków, 259
 dane, 253
 style CSS, 281
 tablice, 146
 tekst z pola tekstowego, 258
 wartości liczbowe, 91
 wyrażenia, 97
przyciski, 222
przypisanie, 35, 46
przypisanie atrybutów stylu CSS, 284
przypisanie stylu CSS, 240
push(), 150, 151

R

random(), 103
RangeError, 169, 170
reagowanie na kliknięcie, 211
reagowanie na ruchy myszy, 213
ReferenceError, 169, 170
referrer, 182
RegExp, 286
 exec(), 299
 metody, 287
 właściwości, 287
regular expressions, 286
releaseEvents(), 176
reload(), 186
removeAttribute(), 240, 243
removeChild(), 199, 201
replace(), 186, 259, 262, 305
 znaki specjalne, 263
reset(), 271
resizable, 180
resizeBy(), 176
resizeTo(), 176

return, 81, 89
reverse(), 150, 151
RGB, 354
rightContext, 287
round(), 103, 105
routeEvent(), 176
rozpoznanie typu przeglądarki, 189
rozszerzone pola tekstowe, 232
równania kwadratowe, 57, 278
ruchy myszy, 213

S

scroll(), 176
scrollbars, 173, 180
scrollBy(), 176
scrollTo(), 176
search, 185
search(), 259, 263, 305
sekwencje ucieczki, 290
sekwencje znaków specjalnych, 25
selectedIndex, 238
self, 173
setAttribute(), 240, 243
setDate(), 329
setFullYear(), 329
setHours(), 329
setInterval(), 176, 342, 346, 348, 353
setMilliseconds(), 329
setMinutes(), 330
setMonth(), 330
setSeconds(), 330
setTime(), 330
setTimeout(), 176, 342, 349
setYear(), 330
shift(), 150, 151
sin(), 103
skrypty, 9
 osadzone, 12
 zewnętrzne, 12
slice(), 150, 260, 263
słowa zarezerwowane, 31
small(), 257
sort(), 150, 153, 154
sortowanie, 153
 specjalne sortowanie, 155
source, 287
spacje, 30
splice(), 150
split(), 260, 264, 305
sprawdzanie poprawności danych, 273
sqrt(), 103, 105

SQRT1_2, 101
SQRT2, 101
stałe matematyczne, 101
standardy JavaScript, 17
standaryzacja modelu DOM, 171
startTimer(), 351
status, 173, 180
statusbar, 173
stop(), 176
strike(), 257
string, 24
 length, 253
string constant, 24
strona HTML, 10
struktura dokumentu (X)HTML, 191
struktura leksykalna, 30
style, 244, 281, 284
style CSS, 201, 202, 240
 class, 249
 className, 249
 dostęp do atrybutów, 240
 dynamiczna zmiana stylów, 248
 manipulowanie stylami, 241
 przypisanie stylu, 240
 style, 244
 usuwanie stylu, 240
styleSheets, 182
sub(), 257
submit(), 271
substr(), 260, 264
substring(), 260, 264
suma bitowa, 44
suma logiczna, 42
sup(), 257
switch, 60
 break, 60, 61
 case, 60
 default, 60
sygnalizacja błędów, 163
symulacja działania metody setInterval(), 349
SyntaxError, 169, 170
systemLanguage, 188

Ś

średniki, 21, 31

T

tabIndex, 221
tablice, 138
 Array, 140
 długość, 144

tablice

dodawanie elementów, 151
 elementy niezdefiniowane, 140
 for...in, 148
 indeksowanie, 141
 indeksy, 144
 length, 144
 liczba komórek, 144
 łączenie tablic, 149
 nielinowa indeksacja, 147
 odczyt danych, 141
 odwracanie kolejności elementów, 151
 operacje, 149
 pętle, 145
 przetwarzanie, 146
 sortowanie, 153
 sortowanie specjalne, 155
 tworzenie, 139, 141
 usuwanie elementów, 151
 właściwości, 144
 zapis danych, 141, 143

tabulatory, 30

tan(), 103
 target, 270
 test(), 287
 testowanie wyrażeń regularnych, 300
 text node, 196

text/ecmascript, 12

text/javascript, 11

this, 116, 127, 134

throw, 157

timery, 342

obsługa, 342

symulacja działania metody setInterval(), 349

uruchamianie licznika, 342

wywołania cykliczne, 346

zatrzymanie zegara, 345

title, 182

tło zależne od pory dnia, 336

toDateString(), 330, 331

toGMTString(), 313, 330, 331

toLocaleDateString(), 331

toLocaleString(), 331

toLocaleTimeString(), 331

toLowerCase(), 260

toolbar, 174, 180

top, 174, 179

top-level objects, 171

toString(), 150, 186, 331

toTimeString(), 331

toUTCString(), 331

true, 26

try, 159

try...catch, 159

zagnieżdżanie bloków, 169

try...catch...finally, 166

tworzenie

elementy strony, 198

funkcje, 78

obiekty, 106, 129

okna, 179

skrypty, 9

tablice, 139

zmienne, 22

TypeError, 169, 170

typeof, 48

typy danych, 19, 23

liczby, 24

logiczny, 26

łańcuchy, 24

obiekty, 26

specjalne, 26

U

ukrywanie kodu przed przeglądarką, 29

umieszczanie skryptów w kodzie (X)HTML, 9, 11

undefined, 26, 87, 100, 140, 147

unescape(), 97

Unicode, 31

unshift(), 150, 151, 152

URI, 92

URIError, 169, 170

URL, 96, 182

userAgent, 188

userLanguage, 188

usuwanie

elementy strony, 199

style CSS, 240

UTC, 328

UTF-8, 10

V

var, 22

vendor, 188

vendorSub, 188

vLink, 182

W

W3C, 171

validacja danych z formularza, 276

wartości logiczne, 26

warunki, 58

wersje JavaScript, 18

węzły, 192, 196
potomne, 197
tekstowe, 192
while, 66
Wide Web Consortium, 171
widoczność zmiennych, 82
width, 179, 182, 244, 250
wielkość liter, 30
 wielokrotne wyświetlanie napisu, 65
window, 172, 174
 metody, 175
 odczyt właściwości, 174
 właściwości, 173
właściwości funkcji, 125
właściwości globalne, 100
wprowadzanie danych, 177, 178, 269, 278
formularze, 269
sprawdzanie poprawności danych, 273
wyrażenia regularne, 307
wrażenia regularne, 307
write(), 184
writeln(), 184
współpraca z przeglądarką, 171
wyjątki, 157
 blok finally, 165
 Error, 158, 160
 nieobsłużone, 158
 nieprzechwycone, 158
 obsługa, 161
 predefiniowane obiekty, 169
 propagacja wyjątków, 169
 przechwytywanie, 159
 try...catch, 159
 zagnieżdżanie bloków try...catch, 169
 zgłoszanie, 157
wykonywanie operacji, 33
wyrażenia regularne, 262, 286
 alternatywy, 297
 atrybuty, 289
 exec(), 299
 flagi, 289
 grupowanie, 295
 klasy znakowe, 291
 kod pocztowy, 308
 kotwice, 298
 metaznaki, 290
 metody, 299
 odwołania w grupowaniu, 295
 powtórzenia, 292
 referencje w grupowaniu, 295
 RegExp, 286
 sekwencje ucieczki, 290
 tworzenie, 288

typ łańcuchowy, 305
weryfikacja danych, 309
wzorzec, 288
zachłanność, 293
zakresy, 292
znaczniki, 289
znaki, 290
wyrażenia warunkowe, 58
wyświetlanie
 dane, 32, 35
 napisy, 22
wywołania cykliczne, 346
wywołanie
 funkcje, 79
 metody, 118

X

XHTML, 11
XOR, 44, 45

Z

zagnieżdżanie
 blok try...catch, 169
 instrukcje warunkowe, 56
 obiekty, 128
 pętle, 70
zaokrąglanie liczb, 105
zapisywane cookies, 317
zasięg zmiennych, 82
zdarzenia, 204
 dynamiczne elementy dokumentu, 218
 dynamiczne przypisywanie
 procedur obsługi, 215
ładowanie strony, 207
model DOM, 205
model obsługi, 205
obsługa, 204
onabort, 206
onblur, 206
onchange, 206
onclick, 206, 211, 223
ondblclick, 206
onerror, 206
onfocus, 206
onkeydown, 206
onkeypress, 206
onkeyup, 206
onload, 207
onmousedown, 207
onmousemove, 207
onmouseout, 207, 213, 215

zdarzenia

onmouseover, 207, 213, 215
onmouseup, 207
onreset, 207
onresize, 207
onselect, 207
onsubmit, 207
onunload, 207, 210
procedura obsługi, 204, 209
zegary, 350
 zestaw znaków ASCII, 31
 zgłaszanie wyjątków, 157
 zliczanie liczby odwiedzin, 323
 zmiana

style CSS, 246
 treść akapitów tekstowych, 214
 zawartość elementu strony, 197
 zawartość warstwy, 211

zmienna iteracyjna pętli, 65

zmienne, 19, 22
 globalne, 83, 128
 lokalne, 83, 84
 nazwy, 23
 typy danych, 26
 wartość początkowa, 27
 zasięg, 82
znaczniki HTML, 27
znaki niestandardowe, 31
znaki specjalne, 25
zwracanie wartości z funkcji, 81

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

- 
- An illustration showing several hands reaching towards a set of interlocking puzzle pieces. One hand has red-painted fingernails. The puzzle pieces are colored brown, tan, and red.
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

Notatki
