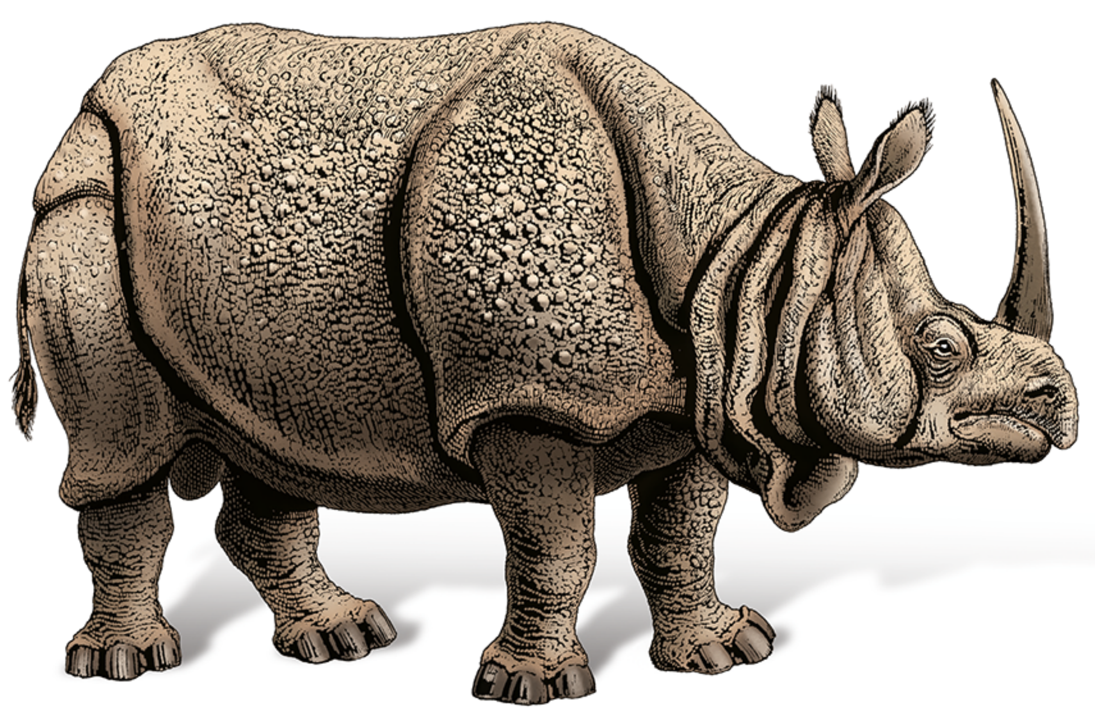
Wydanie VII

JavaScript

Przewodnik

Poznaj język mistrzów programowania



David Flanagan

Tytuł oryginału: JavaScript: The Definitive Guide: Master the World's Most-Used
Programming Language, 7th Edition

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-8322-758-0

© 2021, 2023 Helion S.A.
Authorized Polish translation of the English edition of *JavaScript: The Definitive Guide, 7th Edition*
ISBN 9781491952023 © 2020 David Flanagan

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

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

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

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

Autor oraz wydawca 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 wydawca nie ponoszą również
żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/jspp7v
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/jspp7v.zip

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wstęp 13

1. Wprowadzenie do języka JavaScript 15
1.1. Poznawanie JavaScriptu 17
1.2. Witaj, świecie! 18
1.3. Wycieczka po języku JavaScript 18
1.4. Przykład: histogram częstości użycia znaków 24
1.5. Podsumowanie 26

2. Struktura leksykalna 27
2.1. Tekst programu 27
2.2. Komentarze 27
2.3. Literały 28
2.4. Identyfikatory i zarezerwowane słowa 28
2.5. Unicode 29
2.6. Opcjonalne średniki 30
2.7. Podsumowanie 32

3. Typy, wartości i zmienne 33
3.1. Informacje ogólne i definicje 33
3.2. Liczby 35
3.3. Tekst 41
3.4. Wartości logiczne 47
3.5. Wartości null i undefined 48
3.6. Symbole 49
3.7. Obiekt globalny 50
3.8. Niemutowalne prymitywne wartości i mutowalne odwołania do obiektu 51
3.9. Konwersje typów 52
3.10. Deklarowanie zmiennych i przypisywanie wartości 59
3.11. Podsumowanie 66

Kup ksi k

Pole ksi k

4. Wyrażenia i operatory ... 67
4.1. Wyrażenia podstawowe 67
4.2. Inicjatory obiektów i tablic 68
4.3. Wyrażenia definiujące funkcje 69
4.4. Wyrażenia dostępu do właściwości 69
4.5. Wyrażenia wywołujące 71
4.6. Wyrażenia tworzące obiekty 73
4.7. Przegląd operatorów 74
4.8. Operatory arytmetyczne 78
4.9. Wyrażenia relacyjne 83
4.10. Wyrażenia logiczne 88
4.11. Wyrażenia przypisujące 90
4.12. Wyrażenia interpretujące 92
4.13. Inne operatory 94
4.14. Podsumowanie 99

5. Instrukcje ... 101
5.1. Instrukcje wyrażeniowe 102
5.2. Instrukcje złożone i puste 102
5.3. Instrukcje warunkowe 103
5.4. Pętle 108
5.5. Skoki 114
5.6. Inne instrukcje 121
5.7. Deklaracje 124
5.8. Podsumowanie instrukcji 126

6. Obiekty ... 129
6.1. Wprowadzenie do obiektów 129
6.2. Tworzenie obiektów 130
6.3. Odpytywanie i ustawianie właściwości 132
6.4. Usuwanie właściwości 137
6.5. Sprawdzanie właściwości 138
6.6. Wyliczanie właściwości 139
6.7. Rozszerzanie obiektów 140
6.8. Serializacja obiektów 142
6.9. Metody obiektów 142
6.10. Udoskonalona składnia literału obiektowego 144
6.11. Podsumowanie 150

Kup ksi k Pole ksi k

7. Tablice 151
7.1. Tworzenie tablic 152
7.2. Odczytywanie i zapisywanie elementów tablicy 154
7.3. Rozrzedzone tablice 155
7.4. Długość tablicy 156
7.5. Dodawanie i usuwanie elementów tablicy 156
7.6. Iterowanie tablic 157
7.7. Tablice wielowymiarowe 159
7.8. Metody tablicowe 159
7.9. Obiekty podobne do tablic 170
7.10. Ciągi znaków jako tablice 172
7.11. Podsumowanie 172

8. Funkcje 175
8.1. Definiowanie funkcji 175
8.2. Wywoływanie funkcji 179
8.3. Argumenty i parametry funkcji 185
8.4. Funkcje jako wartości 192
8.5. Funkcje jako przestrzenie nazw 194
8.6. Domknięcia 195
8.7. Właściwości, metody i konstruktory funkcji 199
8.8. Programowanie funkcyjne 203
8.9. Podsumowanie 208

9. Klasy 209
9.1. Klasy i prototypy 210
9.2. Klasy i konstruktory 211
9.3. Słowo kluczowe class 215
9.4. Dodawanie metod do istniejących klas 222
9.5. Podklasy 222
9.6. Podsumowanie 232

10. Moduły 235
10.1. Tworzenie modułów za pomocą klas, obiektów i domknięć 235
10.2. Moduły w środowisku Node 238
10.3. Moduły w języku ES6 240
10.4. Podsumowanie 249

Kup ksi k Pole ksi k

11. Standardowa biblioteka JavaScript 251
11.1. Zbiory i mapy 252
11.2. Typowane tablice i dane binarne 257
11.3. Wyszukiwanie wzorców i wyrażenia regularne 263
11.4. Daty i czas 280
11.5. Klasy błędów 284
11.6. Format JSON, serializacja i analiza składni 285
11.7. Internacjonalizacja aplikacji 288
11.8. Interfejs API konsoli 294
11.9. Interfejs API klasy URL 297
11.10. Czasomierze 300
11.11. Podsumowanie 301

12. Iteratory i generatory 303
12.1. Jak działają iteratory? 304
12.2. Implementowanie obiektów iterowalnych 304
12.3. Generatory 307
12.4. Zaawansowane funkcjonalności generatorów 311
12.5. Podsumowanie 314

13. Asynchroniczność w języku JavaScript 315
13.1. Programowanie asynchroniczne i funkcje zwrotne 315
13.2. Promesy 319
13.3. Słowa kluczowe async i await 338
13.4. Iteracje asynchroniczne 341
13.5. Podsumowanie 346

14. Metaprogramowanie 349
14.1. Atrybuty właściwości 349
14.2. Rozszerzalność obiektów 353
14.3. Atrybut prototype 354
14.4. Popularne symbole 356
14.5. Znaczniki szablonowe 363
14.6. Obiekt Reflect 365
14.7. Klasa Proxy 367
14.8. Podsumowanie 373

Kup ksi k Pole ksi k

15. **JavaScript w przeglądarkach** 375
15.1. Podstawy programowania stron WWW 377
15.2. Zdarzenia 390
15.3. Przetwarzanie dokumentów 399
15.4. Przetwarzanie arkusza stylów 413
15.5. Geometria i przewijanie dokumentu 419
15.6. Komponenty WWW 423
15.7. SVG: skalowalna grafika wektorowa 434
15.8. Grafika w znaczniku <canvas> 440
15.9. Klasa Audio 460
15.10. Lokalizacja, nawigacja i historia 462
15.11. Sieć 470
15.12. Magazynowanie danych 485
15.13. Wątki robocze i komunikaty 496
15.14. Przykład: zbiór Mandelbrota 503
15.15. Podsumowanie i dalsza lektura 514

16. **Serwery w środowisku Node** 521
16.1. Podstawy programowania w środowisku Node 522
16.2. Domyślna asynchroniczność 526
16.3. Bufory 529
16.4. Zdarzenia i klasa EventEmitter 531
16.5. Strumienie 533
16.6. Procesy, procesory i szczegóły systemu operacyjnego 543
16.7. Operacje na plikach 544
16.8. Klienty i serwery HTTP 554
16.9. Klienty i serwery inne niż HTTP 558
16.10. Procesy potomne 560
16.11. Wątki robocze 565
16.12. Podsumowanie 573

17. **Narzędzia i rozszerzenia** 575
17.1. Inspekcja kodu za pomocą narzędzia ESLint 575
17.2. Formatowanie kodu za pomocą narzędzia Prettier 576
17.3. Tworzenie testów jednostkowych za pomocą narzędzia Jest 577
17.4. Zarządzanie pakietami za pomocą narzędzia npm 580
17.5. Pakowanie kodu 581
17.6. Transpilacja kodu za pomocą narzędzia Babel 583
17.7. Rozszerzenie JSX: znaczniki w kodzie JavaScript 584
17.8. Sprawdzanie typów danych za pomocą rozszerzenia Flow 588
17.9. Podsumowanie 602

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

ROZDZIAŁ 4.

Wyrażenia i operatory

W tym rozdziale opisane są wyrażenia stosowane w języku JavaScript oraz wykorzystywane w nich operatory. **Wyrażenie** to fraza, którą można wyliczyć i uzyskać wartość. Bardzo prostym przykładem wyrażenia jest stała. Zmienna też jest wyrażeniem, którego wynikiem jest wartość przypisana tej zmiennej. Złożone wyrażenia składają się z prostszych wyrażeń. Na przykład odwołanie do tablicy składa się z wyrażenia reprezentującego tablicę, kwadratowego nawiasu otwierającego, drugiego wyrażenia (którego wartością jest liczba całkowita) oraz zamykającego nawiasu kwadratowego. Wynikiem utworzonego w ten sposób nowego, bardziej złożonego wyrażenia jest wartość elementu zadanej tablicy o zadanym indeksie. Podobnie wyrażenie wywołujące funkcję składa się z wyrażenia, którego wynikiem jest obiekt reprezentujący daną funkcję, oraz kilku ewentualnych dodatkowych wyrażeń będących jej argumentami.

Złożone wyrażenia najczęściej tworzy się za pomocą prostszych wyrażeń i **operatorów**. Operator łączy w określony sposób wartości **operandów** (zazwyczaj dwóch) i tworzy nową wartość. Prostym przykładem jest operator mnożenia `*`. Wynikiem wyrażenia `x * y` jest iloczyn wartości zmiennych `x` i `y`. Czasami mówi się, upraszczając, że operator *zwraca* wartość.

W tym rozdziale opisane są wszystkie operatory stosowane w języku JavaScript, jak również wyrażenia, które nie wykorzystują operatorów (na przykład indeksy tablicy czy wywołanie funkcji). Jeżeli znasz inny język programowania o składni podobnej do stosowanej w języku C, przekonasz się, że składnia większości wyrażeń JavaScript i operatorów jest podobna.

4.1. Wyrażenia podstawowe

Najprostsze są **wyrażenia podstawowe**, tzn. takie, które nie składają się z jeszcze prostszych wyrażeń. Wyrażeniami podstawowymi w języku JavaScript są stałe, **literały** wartości, niektóre słowa kluczowe i odwołania do zmiennych.

Literały to stałe wartości wpisane bezpośrednio w kodzie programu, na przykład:

```
1.23          // Literał liczbowy.
"cześć"       // Literał tekstowy.
/szablon/     // Literał wyrażenia regularnego.
```


Kup ksi k   Pole ksi k

Składnia literałów liczbowych została opisana w podrozdziale 3.2, a tekstowych w podrozdziale 3.3. Literały wyrażeń regularnych zostały ogólnie przedstawione w punkcie 3.3.5, natomiast szczegółowo będą opisane w podrozdziale 11.3.

Wyrażeniami podstawowymi są również niektóre zarezerwowane słowa:

```
true        // Logiczna wartość "prawda".
false       // Logiczna wartość "fałsz".
null        // "Pusta" wartość.
this        // Wartość oznaczająca "bieżący" obiekt.
```

W podrozdziałach 3.4 i 3.5 poznałeś słowa `true`, `false` i `null`, które w odróżnieniu od innych słów kluczowych nie są stałymi, ponieważ mogą przyjmować różne wartości w zależności od miejsca, w którym zostaną użyte. Słowo `this` jest stosowane w językach obiektowych. Użyte w ciele metody oznacza obiekt, którego metoda została wywołana. Więcej informacji na ten temat znajdziesz w podrozdziale 4.5, rozdziale 8. (szczególnie w punkcie 8.2.2) i rozdziale 9.

Trzecim rodzajem wyrażenia podstawowego jest odwołanie do zmiennej, stałej lub do właściwości obiektu globalnego:

```
i              // Wynikiem jest wartość zmiennej i.
sum            // Wynikiem jest wartość zmiennej sum.
undefined      // Wynikiem jest wartość właściwości "undefined" obiektu globalnego.
```

Interpreter JavaScript traktuje każdy użyty w kodzie identyfikator jako zmienną, stałą lub właściwość obiektu globalnego i stara się uzyskać jego wartość. Jeżeli z identyfikatorem nie jest skojarzona żadna wartość, próba jej uzyskania powoduje zgłoszenie wyjątku ReferenceError.

4.2. Inicjatory obiektów i tablic

Inicjator obiektu lub **tablicy** jest wyrażeniem, którego wartością jest nowo utworzony obiekt lub tablica. Tego rodzaju wyrażenie jest czasem nazywane **literałem obiektowym** lub **tablicowym**. W odróżnieniu od prawdziwego literału nie jest to jednak wyrażenie podstawowe, jako że zawiera w sobie podwyrażenie określające wartość właściwości obiektu lub elementu tablicy. Ponieważ składnia inicjatora tablicy jest nieco prostsza, przyjrzymy się jej w pierwszej kolejności.

Inicjator tablicy jest listą wyrażeń oddzielonych przecinkami, umieszczoną wewnątrz nawiasów kwadratowych. Wynikiem inicjatora jest nowa tablica elementów zainicjowanych wynikami oddzielonych przecinkami wyrażeń:

```
[]          // Pusta tablica. Brak wyrażeń wewnątrz nawiasów oznacza, że tablica nie ma elementów.
[1+2,3+4]   // Tablica dwuelementowa. Pierwszy element ma wartość 3, a drugi 7.
```

Poszczególne wyrażenia inicjatora tablicy mogą być inicjatorami innych tablic. W ten sposób można tworzyć zagnieżdżone tablice:

```
let matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Wyniki wyrażeń inicjujących elementy są wyliczane za każdym razem, gdy wyliczany jest wynik samego inicjatora. Oznacza to, że za każdym razem wynik ten może być inny.

Kup ksi k Pole ksi k

Jeżeli w literale tablicowym pominie się wartości rozdzielone przecinkami, wówczas elementy te nie zostaną zdefiniowane. Na przykład poniższa tablica składa się z pięciu elementów, z których trzy są niezdefiniowane:

```
let sparseArray = [1,,,,5];
```

Jeżeli w inicjatorze tablicy po ostatnim wyrażeniu umieści się przecinek, nie spowoduje on utworzenia niezdefiniowanego elementu. Jednak wynikiem wyrażenia odwołującego się do elementu poza ostatnim wyrażeniem inicjującym będzie wartość `undefined`.

Wyrażenie inicjujące obiekt jest podobne do wyrażenia inicjującego tablicę. Różnica polega na tym, że zamiast nawiasów kwadratowych stosuje się nawiasy klamrowe, a każde podwyrażenie jest poprzedzone nazwą właściwości i dwukropkiem:

```
let p = { x: 2.3, y: -1.2 };  // Obiekt posiadający dwie właściwości.
let q = {};                   // Obiekt bez właściwości.
q.x = 2.3; q.y = -1.2;        // Obiekt q ma teraz te same właściwości co obiekt p.
```

Począwszy od wersji ES6 języka literały obiektowe mają znacznie bogatszą składnię (szczegółowe informacje znajdziesz w podrozdziale 6.10). Na przykład można je zagnieżdżać:

```
let rectangle = {
  upperLeft: { x: 2, y: 2 },
  lowerRight: { x: 4, y: 5 }
};
```

Z inicjatorami obiektów i tablic spotkasz się jeszcze w rozdziałach 6. i 7.

4.3. Wyrażenia definiujące funkcje

Wynikiem **wyrażenia definiującego funkcję** jest nowa funkcja. Tego rodzaju wyrażenie jest literałem funkcyjnym, podobnie jak inicjator obiektu jest literałem obiektowym. Wyrażenie definiujące funkcję zazwyczaj składa się ze słowa kluczowego `function`, następujących po nim nawiasów zwykłych, wewnątrz których można umieścić listę oddzielonych przecinkami identyfikatorów (nazw parametrów), oraz bloku kodu (ciała funkcji) zamkniętego w nawiasach klamrowych. Poniżej przedstawiony jest przykład:

```
// Funkcja zwracająca kwadrat argumentu.
let square = function(x) { return x * x; };
```

Wyrażenie definiujące funkcję może również zawierać jej nazwę. Funkcję można też definiować za pomocą instrukcji, a nie wyrażenia. W wersjach języka ES6 i nowszych można stosować nową, zwięzłą składnię „strzałkową”. Szczegółowe informacje na temat definiowania funkcji znajdziesz w rozdziale 8.

4.4. Wyrażenia dostępu do właściwości

Wynikiem **wyrażenia dostępu do właściwości** jest wartość właściwości obiektu lub elementu tablicy. Język JavaScript definiuje dwie składnie dostępu do właściwości:

```
wyrażenie . identyfikator
wyrażenie [ wyrażenie]
```


Kup ksi k   Pole ksi k

Pierwsza składnia to wyrażenie z następującą po nim kropką i identyfikatorem. Wyrażenie określa obiekt, a identyfikator nazwę żądanej właściwości. Druga składnia to wyrażenie (obiekt lub tablica) z następującym po nim drugim wyrażeniem umieszczonym wewnątrz nawiasów kwadratowych. Wyrażenie to określa nazwę żądanej właściwości lub indeks żądanego elementu tablicy. Poniżej przedstawionych jest kilka przykładów:

```
let o = {x: 1, y: {z: 3}}; // Przykładowy obiekt.
let a = [o, 4, [5, 6]];    // Przykładowa tablica zawierająca obiekt.
o.x                        // => 1: właściwość x wyrażenia o.
o.y.z                      // => 3: właściwość z wyrażenia o.y.
o["x"]                     // => 1: właściwość x obiektu o.
a[1]                       // => 4: element o indeksie 1 wyrażenia a.
a[2]["1"]                  // => 6: element o indeksie 1 wyrażenia a[2].
a[0].x                     // => 1: właściwość x wyrażenia a[0].
```

W obu składniach najpierw wyliczany jest wynik wyrażenia umieszczonego przed kropką lub otwierającym nawiasem kwadratowym. Jeżeli wynikiem jest wartość `null` lub `undefined`, zgłaszany jest wyjątek TypeError, ponieważ żadna z tych wartości nie ma właściwości. Jeżeli po wyrażeniu obiektu znajduje się kropka i identyfikator, odczytywana jest wartość właściwości o nazwie takiej jak identyfikator, która staje się wynikiem całego wyrażenia. Jeżeli po wyrażeniu obiektu znajduje się inne wyrażenie umieszczone wewnątrz nawiasów kwadratowych, wyliczany jest jego wynik, przekształcany następnie w ciąg znaków. Ostatecznym wynikiem wyrażenia jest wartość właściwości o nazwie takiej jak uzyskany ciąg. W obu składniach, w przypadku braku właściwości o zadanej nazwie, wynikiem wyrażenia dostępu do właściwości jest `undefined`.

Składnia z identyfikatorem jest prostsza. Zwróć jednak uwagę, że można ją stosować jedynie wtedy, gdy nazwa żądanej właściwości jest poprawnym identyfikatorem i jest znana w chwili pisania kodu. Jeżeli nazwa właściwości zawiera spacje lub znaki specjalne albo jest liczbą (w przypadku tablicy), należy użyć składni z nawiasami kwadratowymi. Nawiasy są również stosowane wtedy, gdy nazwa właściwości nie jest statyczna, czyli jest wynikiem wyliczeń (patrz przykład w punkcie 6.3.1).

Obiekty i ich właściwości będą szczegółowo opisane w rozdziale 6., a tablice i ich elementy w rozdziale 7.

4.4.1. Warunkowy dostęp do właściwości

W wersji ES2020 zostały wprowadzone dwa nowe rodzaje wyrażeń dostępu do właściwości:

```
wyrażenie ?. identyfikator
wyrażenie ?.[ wyrażenie ]
```

Wartości `null` i `undefined` są jedynymi wartościami w języku JavaScript, które nie mają właściwości. W ich przypadku próba odwołania się do właściwości za pomocą kropki lub nawiasów `[]` skutkuje zgłoszeniem wyjątku TypeError. Aby uchronić się przed takimi sytuacjami, można użyć notacji `?.` lub `?.[]`.

Przeanalizujmy wyrażenie `a?.b`. Jeżeli zmienna `a` ma wartość `null` lub `undefined`, wynikiem całego wyrażenia jest `undefined`. Nie trzeba przy tym odwoływać się do właściwości `b`. Jeżeli zmienna `a` ma inną wartość, wynikiem całego wyrażenia jest wartość właściwości `a.b` (jeżeli `a` nie ma właściwości o nazwie `b`, wynikiem wyrażenia również jest `undefined`).

Kup ksi k Pole ksi k

Tego rodzaju wyrażenie dostępu do właściwości jest niekiedy nazywane „opcjonalnym łańcuchowaniem”, ponieważ może składać się z większej liczby połączonych wyrażeń, na przykład:

```
let a = { b: null };
a.b?.c.d    // => undefined
```

W powyższym przykładzie `a` jest obiektem, więc `a.b` jest poprawnym wyrażeniem dostępu do właściwości. Jednak właściwość `a.b` ma wartość `null`, więc użycie wyrażenia `a.b.c` skutkuje zgłoszeniem wyjątku TypeError. Jeżeli zamiast kropki użyje się zapisu `?.`, wyjątek nie zostanie zgłoszony, a wynikiem wyrażenia `a.b?.c` będzie `undefined`. Oznacza to, że wyrażenie `(a.b?.c).d` zgłosi wyjątek TypeError, ponieważ odwołuje się do właściwości wartości `undefined`. Natomiast wynikiem wyrażenia `a.b?.c.d` (bez nawiasów) jest wartość `undefined` i błąd nie jest zgłaszany. Ujawnia się tu bardzo ważna cecha opcjonalnego łańcuchowania, zwana „krótkim zwarciem”: jeżeli wynikiem podwyrażenia znajdującego się po lewej stronie sekwencji `?.` jest wartość `null` lub `undefined`, wówczas jako wynik całego wyrażenia jest przyjmowana wartość `undefined` bez uprzedniego odwoływania się do właściwości umieszczonej po prawej stronie.

Oczywiście, jeżeli wynikiem wyrażenia `a.b` jest obiekt, który nie posiada właściwości o nazwie `c`, również zgłaszany jest wyjątek TypeError. W takim przypadku trzeba użyć kolejnego warunkowego wyrażenia dostępu do właściwości:

```
let a = { b: {} };
a.b?.c?.d   // => undefined
```

W wyrażeniu warunkowego dostępu do właściwości można również stosować zapis `?.[]` zamiast `[]`. Przeanalizujmy wyrażenie `a?.[b][c]`. Jeżeli zmienna `a` ma wartość `null` lub `undefined`, wówczas jako wynik całego wyrażenia jest natychmiast przyjmowana wartość `undefined` bez uprzedniego sprawdzania podwyrażeń `b` i `c`. Jeżeli wyrażenia te wywołują jakieś skutki uboczne, nie będą wtedy miały miejsca:

```
let a;          // Ups, zapomnieliśmy zainicjować tę zmienną!
let index = 0;
try {
    a[index++]; // Zgłoszenie wyjątku TypeError.
} catch(e) {
    index       // => 1: zmienna jest powiększana przed zgłoszeniem wyjątku TypeError.
}
a?.[index++]    // => undefined: ponieważ zmienna a ma wartość undefined.
index           // => 1: zmienna nie jest powiększana, ponieważ zapis ?.[] powoduje krótkie zwarcie.
a[index++]      // TypeError: nie można indeksować niezdefiniowanej zmiennej.
```

Warunkowy dostęp do właściwości przy użyciu notacji `?.` i `?.[]` jest jedną z najnowszych funkcjonalności języka JavaScript. Na początku 2020 r. składnia ta była obsługiwana we wstępnych wersjach większości najpopularniejszych przeglądarek.

4.5. Wyrażenia wywołujące

Wyrażenie wywołujące jest frazą powodującą wywołanie (uruchomienie) funkcji lub metody. Obejmuje ona wyrażenie identyfikujące funkcję, która ma być wywołana, z następującymi po niej nawiasami zwykłymi, wewnątrz których może znajdować się lista rozdzielonych przecinkami wyrażeń argumentów. Poniżej przedstawionych jest kilka przykładów:

Kup ksi k   Pole ksi k

```
f(0)             // f jest wyrażeniem funkcyjnym, a 0 wyrażeniem argumentu.
Math.max(x,y,z)  // Math.max jest funkcją, a x, y, i z są argumentami.
a.sort()         // a.sort jest funkcją bez argumentów.
```

Podczas wyliczania wyrażenia wywołującego najpierw wyliczany jest wynik wyrażenia funkcyjnego. Następnie wyliczane są wyniki wyrażeń argumentów i tworzona lista wartości. Jeżeli wynikiem wyrażenia funkcyjnego nie jest funkcja, zgłaszany jest wyjątek TypeError. W przeciwnym razie parametrom określonym w definicji funkcji są przypisywane wartości argumentów i na koniec wykonywany jest kod funkcji. Jeżeli w kodzie użyta jest instrukcja `return`, zwracana przez nią wartość staje się wynikiem całego wyrażenia wywołującego. W rozdziale 8. będą opisane wszystkie szczegóły dotyczące wywoływania funkcji oraz operacje wykonywane w sytuacji, gdy liczba wyrażeń argumentów różni się od liczby parametrów w definicji funkcji.

Każde wyrażenie wywołujące zawiera parę nawiasów i podwyrażenie umieszczone przed nawiasem otwierającym. Jeżeli jest to wyrażenie dostępu do właściwości, wówczas mamy do czynienia z **wywołaniem metody**. W takim przypadku obiekt lub tablica, której dotyczy odwołanie, po uruchomianiu kodu funkcji staje się wartością słowa kluczowego `this`. Jest to paradygmat programowania obiektowego, w którym funkcje (nazywane „metodami”, jeżeli są wywoływane w opisany sposób) działają na obiekcie, którego są częściami. Szczegółowe informacje na ten temat znajdziesz w rozdziale 9.

4.5.1. Wywołania warunkowe

W wersji języka ES2020 można wywoływać funkcje, stosując notację `?.()` zamiast `()`. Jeżeli funkcja jest wywoływana w zwykły sposób, a wyrażenie znajdujące się przed nawiasami ma wartość `null`, `undefined` lub nie jest funkcją, jest zgłaszany wyjątek TypeError. Jeżeli natomiast użyje się nowej notacji `?.()` i wyrażenie po lewej stronie znaku zapytania będzie miało wartość `null` lub `undefined`, to wynikiem całego wyrażenia wywołującego będzie wartość `undefined`, a wyjątek nie zostanie zgłoszony.

Klasa `Array` posiada metodę, w której opcjonalnym argumencie można umieszczać funkcję definiującą porządek sortowania. W wersjach starszych niż ES2020, tworząc metodę z opcjonalnym argumentem funkcyjnym, taką jak `sort()`, trzeba było przed wywołaniem argumentu funkcyjnego sprawdzać za pomocą instrukcji `if`, czy został on określony:

```
function square(x, log) { // Drugim argumentem jest opcjonalna funkcja.
  if (log) {              // Jeżeli opcjonalny argument został określony,
    log(x);               // wywołaj go.
  }
  return x * x;           // Zwrócenie kwadratu argumentu.
}
```

Począwszy od wersji ES2020, stosując notację wywołania warunkowego `?.()`, można uprościć kod. Oczywiście wywołanie będzie miało miejsce tylko wtedy, gdy argument będzie funkcją:

```
function square(x, log) { // Drugim argumentem jest opcjonalna funkcja.
  log?.(x);               // Wywołanie funkcji, jeżeli została określona.
  return x * x;           // Zwrócenie kwadratu argumentu.
}
```


Kup ksi k Pole ksi k

Zwróć uwagę, że notacja `?.()` sprawdza jedynie, czy po lewej stronie znajduje się wartość `null` lub `undefined`. Nie sprawdza, czy wartość jest funkcją, którą można wywołać. Zatem w powyższym przykładzie funkcja `square()` zgłosi wyjątek, jeżeli w jej argumentach zostaną umieszczone na przykład dwie liczby.

W warunkowym wywołaniu funkcyjnym `?.()`, podobnie jak w warunkowym wyrażeniu dostępu do właściwości (patrz punkt 4.4.1), obowiązuje reguła krótkiego zwarcia. Jeżeli wartość umieszczona po lewej stronie znaku zapytania jest równa `null` lub `undefined`, wówczas nie jest wyliczane żadne z wyrażeń umieszczonych wewnątrz nawiasów:

```
let f = null, x = 0;
try {
  f(x++);     // Zgłoszenie wyjątku TypeError, jeżeli f ma wartość null.
} catch(e) {
  X           // => 1: zmienna x jest zwiększana przed zgłoszeniem wyjątku TypeError.
}
f?.(x++)      // => undefined: zmienna f ma wartość null, ale wyjątek nie jest zgłaszany.
x             // => 1: instrukcja zwiększająca zmienną jest pomijana, zgodnie z regułą krótkiego zwarcia.
```

Wyrażenie wywołania warunkowego `?.()` sprawdza się zarówno w przypadku metod, jak i funkcji. Ponieważ jednak wywołanie jest również odwołaniem do właściwości, upewnij się, czy zrozumiałe są różnice pomiędzy tymi dwiema operacjami:

```
o.m()     // Zwykłe odwołanie do właściwości i zwykłe wywołanie metody.
o?.m()    // Warunkowe odwołanie do właściwości i zwykłe wywołanie metody.
o.m?.()   // Zwykłe odwołanie do właściwości i warunkowe wywołanie metody.
```

W pierwszym wyrażeniu zmienna `o` musi być obiektem posiadającym właściwość `m`, którego wartością musi być funkcja. W drugim przypadku, jeżeli `o` ma wartość `null` lub `undefined`, wynikiem całego wyrażenia jest `undefined`. Jeżeli `o` ma inną wartość, musi posiadać właściwość `m`, którego wartością musi być funkcja. W trzecim przypadku wartością `o` nie może być `null` ani `undefined`. Jeżeli zmienna nie ma właściwości o nazwie `m` lub ma taką właściwość o wartości `null`, wówczas wynikiem całego wyrażenia jest `undefined`.

Warunkowe wywołanie `?.()` jest jedną z najnowszych funkcjonalności języka JavaScript. W pierwszych miesiącach 2020 r. było obsługiwane we wstępnych wersjach większości najpopularniejszych przeglądarek.

4.6. Wyrażenia tworzące obiekty

Wyrażenie tworzące obiekt wywołuje funkcję (tzw. konstruktor) inicjującą właściwości nowo utworzonego obiektu. Tego rodzaju wyrażenia są podobne do wyrażeń wywołujących. Różnią się jednak tym, że poprzedza się je słowem kluczowym `new`:

```
new Object()
new Point(2,3)
```

Jeżeli w argumentach konstruktora nie trzeba umieszczać wartości, można pominąć parę nawiasów:

```
new Object
new Date
```


Kup książkę Poleć książkę

Wynikiem wyrażenia tworzącego obiekt jest nowy obiekt. Konstruktory będą szczegółowo opisane w rozdziale 9.

4.7. Przegląd operatorów

Operatory są stosowane w wyrażeniach arytmetycznych, porównujących, logicznych, przypisujących i innych. Tabela 4.1 zawiera podsumowanie operatorów i stanowi podręczne odniesienie.

Tabela 4.1. Operatory w języku JavaScript

Operator	Operacja	W	L	Typy
`++`	Pre- i postinkrementacja	P	1	*l*-wartość → liczba
`--`	Pre- i postdekrementacja	P	1	*l*-wartość → liczba
`-`	Zmiana znaku liczby	P	1	liczba → liczba
`+`	Konwersja na liczbę	P	1	dowolny → liczba
`~`	Odwrócenie bitów	P	1	liczba całkowita → liczba całkowita
`!`	Negacja wartości logicznej	P	1	wart. logiczna → wart. logiczna
`delete`	Usunięcie właściwości	P	1	*l*-wartość → wart. logiczna
`typeof`	Określenie typu operandu	P	1	dowolny → ciąg znaków
`void`	Zwrócenie pustej wartości	P	1	dowolny → `undefined`
`**`	Potęgowanie	P	2	liczba, liczba → liczba
`*`, `/`, `%`	Mnożenie, dzielenie, reszta	L	2	liczba, liczba → liczba
`+`, `-`	Dodawanie, odejmowanie	L	2	liczba, liczba → liczba
`+`	Łączenie ciągów znaków	L	2	ciąg znaków, ciąg znaków → ciąg znaków
`<<`	Przesunięcie bitów w lewo	L	2	liczba całkowita, liczba całkowita → liczba całkowita
`>>`	Przesunięcie bitów w prawo z zachowaniem znaku	L	2	liczba całkowita, liczba całkowita → liczba całkowita
`>>>`	Przesunięcie bitów w prawo z uzupełnieniem zerami	L	2	liczba całkowita, liczba całkowita → liczba całkowita
`<`, `<=`, `>`, `>=`	Porównanie liczbowe	L	2	liczba, liczba → wart. logiczna
`<`, `<=`, `>`, `>=`	Porównanie alfabetyczne	L	2	ciąg znaków, ciąg znaków → wart. logiczna
`instanceof`	Określenie klasy obiektu	L	2	obiekt, funkcja → wart. logiczna
`in`	Sprawdzenie istnienia właściwości	L	2	dowolny, obiekt → wart. logiczna
`==`	Nieścisła równość	L	2	dowolny, dowolny → wart. logiczna

Kup ksi k Pole ksi k

Tabela 4.1. Operatory w języku JavaScript (ciąg dalszy)

<table>
<tr><th>Operator</th><th>Operacja</th><th>W</th><th>L</th><th>Typy</th></tr>
<tr><td>!=</td><td>Nieścisła nierówność</td><td>L</td><td>2</td><td>dowolny, dowolny → wart. logiczna</td></tr>
<tr><td>===</td><td>Ścisła równość</td><td>L</td><td>2</td><td>dowolny, dowolny → wart. logiczna</td></tr>
<tr><td>!==</td><td>Ścisła nierówność</td><td>L</td><td>2</td><td>dowolny, dowolny → wart. logiczna</td></tr>
<tr><td>&</td><td>Bitowa operacja ORAZ</td><td>L</td><td>2</td><td>liczba całkowita, liczba całkowita → liczba całkowita</td></tr>
<tr><td>^</td><td>Bitowa różnica symetryczna</td><td>L</td><td>2</td><td>liczba całkowita, liczba całkowita → liczba całkowita</td></tr>
<tr><td>|</td><td>Bitowa operacja LUB</td><td>L</td><td>2</td><td>liczba całkowita, liczba całkowita → liczba całkowita</td></tr>
<tr><td>&&</td><td>Logiczna operacja ORAZ</td><td>L</td><td>2</td><td>dowolny, dowolny → dowolny</td></tr>
<tr><td>||</td><td>Logiczna operacja LUB</td><td>L</td><td>2</td><td>dowolny, dowolny → dowolny</td></tr>
<tr><td>??</td><td>Wybranie pierwszego zdefiniowanego operandu</td><td>L</td><td>2</td><td>dowolny, dowolny → dowolny</td></tr>
<tr><td>?:</td><td>Wybranie drugiego lub trzeciego operandu</td><td>P</td><td>3</td><td>wart. logiczna, dowolny, dowolny → dowolny</td></tr>
<tr><td>=</td><td>Przypisanie wartości zmiennej lub właściwości</td><td>P</td><td>2</td><td>l-wartość, dowolny → dowolny</td></tr>
<tr><td>**=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=</td><td>Wykonanie operacji z przypisaniem wartości</td><td>P</td><td>2</td><td>l-wartość, dowolny → dowolny</td></tr>
<tr><td>,</td><td>Pominięcie pierwszego operandu i zwrócenie drugiego</td><td>L</td><td>2</td><td>dowolny, dowolny → dowolny</td></tr>
</table>

Zwróć uwagę, że większość operatorów zapisuje się za pomocą znaków specjalnych, na przykład + lub =. Niektóre jednak są słowami kluczowymi, jak `delete` lub `instanceof`. Tego rodzaju operatory są zwykłymi wyrażeniami, podobnymi do innych wyrażeń o mniej zwięzłej składni i zawierających znaki specjalne.

Operatory w tabeli 4.1 są ułożone według ich priorytetów. Na początku wymienione są operatory o najwyższych priorytetach. Poziome linie oddzielają grupy operatorów o różnych priorytetach. Kolumna *W* określa wiązanie operatora, które może być lewostronne (*L*) lub prawostronne (*P*). Kolumna *L* zawiera liczbę operandów. W kolumnie *Typy* wymienione są typy operandów i typ wyniku operatora (po symbolu →). W kolejnych punktach wyjaśnione są priorytety, wiązania, typy operatorów i poszczególne operatory.

Kup ksi k Pole ksi k

4.7.1. Liczba operandów

Operatory można podzielić na kategorie według liczby operandów (*arności*). Większość operatorów, na przykład mnożenia *, jest **dwuargumentowych**. Łączą one dwa wyrażenia w jedno, bardziej złożone wyrażenie. W języku JavaScript dostępne są również operatory **jednoargumentowe**, przekształcające pojedyncze wyrażenie w inne, bardziej złożone. Operator – w wyrażeniu `-x` jest operatorem jednoargumentowym zmieniającym znak operandu `x`. Jest jeszcze warunkowy operator trójargumentowy `?:`, łączący trzy wyrażenia w jedno.

4.7.2. Typy operandów i wyników

Niektóre operatory działają na wartościach dowolnych typów, jednak większość wymaga, aby ich operandy były określonych typów. Zwracany wynik też ma określony typ. W tabeli 4.1 w kolumnie *Typy* wymienione są typy operandów (przed strzałkami) i typy wyników operatorów (za strzałkami).

Operatory zazwyczaj konwertują typy operandów odpowiednio do wymagań (patrz podrozdział 3.9). Na przykład operator mnożenia * wymaga, aby operandy były liczbami, ale wyrażenie `"3" * "5"` jest poprawne, ponieważ użyte operandy mogą być przekształcone w liczby. W tym przypadku wynikiem wyrażenia jest oczywiście liczba 15, a nie ciąg `"15"`. Pamiętaj, że każda wartość w języku JavaScript jest „prawdziwa” lub „fałszywa”, zatem operatory logiczne działają na wartościach wszystkich typów.

Niektóre operatory działają różnie w zależności od typów operandów. Dotyczy to głównie operatora +, który dodaje operandy liczbowe, jak również łączy operandy tekstowe. Podobnie operatory porównania, na przykład <, stosują porządek liczbowy lub alfabetyczny w zależności od typów operandów. W opisach poszczególnych operatorów można znaleźć wyjaśnienia dotyczące ich działania w zależności od typów operandów oraz rodzaju przeprowadzanych konwersji.

Zwróć uwagę, że operatory przypisania i kilka innych wymagają operandu typu ***l*-wartość**. Jest to historyczny termin oznaczający „poprawne wyrażenie znajdujące się po lewej stronie operatora przypisania”. W języku JavaScript *l*-wartościami są zmienne, właściwości obiektów i elementy tablic.

4.7.3. Efekty uboczne operatorów

Wyliczanie wyniku wyrażenia na przykład `2 * 3` nie wpływa na stan programu ani na wyniki przyszłych wyliczeń. Jednak niektóre wyrażenia wywołują **efekty uboczne** i wpływają na wyniki wyliczeń, które będą wykonane w przyszłości. Najlepszym przykładem jest operator przypisania. Jeżeli zmiennej lub właściwości zostanie przypisana wartość, zmienią się wyniki wyrażeń wykorzystujących tę zmienną lub właściwość. Podobnie jest w przypadku operatorów inkrementacji ++ i dekrementacji —, wykonujących niejawne przypisanie. Operator `delete` również wywołuje efekt uboczny, ponieważ usunięcie właściwości można traktować jako przypisanie jej wartości `undefined` (choć nie jest to dokładnie to samo).

Inne operatory nie wywołują efektów ubocznych. Wyrażenia wywołujące funkcje i tworzące obiekty mogą wywoływać efekty, jeżeli wywołują je operatory użyte w ciele funkcji lub konstruktora.

Kup ksi k Pole ksi k

4.7.4. Priorytety operatorów

Operatory wymienione w tabeli 4.1 są ułożone według priorytetów, od najwyższych do najniższych. Poziome linie oddzielają grupy operatorów o takich samym priorytetach. Priorytety określają kolejność stosowania operatorów. Operator o wyższym priorytecie (znajdującym się bliżej początku tabeli) jest stosowany przed operatorem o priorytecie niższym (umieszczonym bliżej końca tabeli). Przeanalizujmy poniższe wyrażenie:

```
w = x + y*z;
```

Operator mnożenia * ma wyższy priorytet niż operator dodawania +, więc mnożenie jest wykonywane przed dodawaniem. Najniższy priorytet ma operator przypisania =, więc jest stosowany po wyliczeniu wyniku wyrażenia znajdującego się po jego prawej stronie.

Kolejność stosowania operatorów można zmieniać za pomocą nawiasów. Aby w powyższym przykładzie wymusić najpierw wykonanie dodawania, należy wyrażenie sformułować w następujący sposób:

```
w = (x + y)*z;
```

Zwróć uwagę, że odwołanie do właściwości i wywołanie funkcji mają wyższy priorytet niż pozostałe operatory w tabeli 4.1. Przeanalizujmy następujące wyrażenie:

```
// Zmienna my jest obiektem posiadającym właściwość o nazwie functions, której wartością
// jest tablica funkcji. Wywołujemy funkcję o numerze x, umieszczamy w jej
// argumencie wartość y i sprawdzamy typ zwróconego wyniku.
typeof my.functions[x](y)
```

Operator `typeof` ma jeden z najwyższych priorytetów, ale operuje na wynikach wyrażenia dostępu do właściwości, indeksie tablicy i wywołaniu funkcji, dlatego te operacje są wykonywane w pierwszej kolejności.

W praktyce, gdy pojawiają się wątpliwości dotyczące priorytetów operatorów, najlepiej jest użyć nawiasów jawnie określających kolejność operacji. Warto jednak znać następujące ważne zasady: mnożenie i dzielenie jest wykonywane przed dodawaniem i odejmowaniem, a przypisanie ma bardzo niski priorytet i niemal zawsze jest wykonywane na końcu.

Nowe operatory wprowadzone do języka JavaScript nie zawsze pasowały do opisanego schematu priorytetów. Na przykład operator `??` (patrz punkt 4.13.2) ma według tabeli niższy priorytet niż operatory `||` i `&&`, jednak w rzeczywistości względne priorytety tych operatorów nie są określone. Dlatego począwszy od wersji ES2020 w wyrażeniach wykorzystujących powyższe operatory trzeba jawnie określać kolejność ich stosowania za pomocą nawiasów. Podobnie operator potęgowania `**` nie ma ściśle określonego względnego priorytetu wobec jednoargumentowego operatora zmiany znaku liczby. Dlatego w wyrażeniach wykorzystujących oba operatory trzeba stosować nawiasy.

4.7.5. Wiązanie operatorów

W tabeli 4.1 w kolumnie *W* określone są **wiązania** operatorów. Litery *L* i *P* oznaczają, odpowiednio, wiązanie lewostronne i prawostronne. Wiązanie określa kolejność wykonywania operacji o takich

Kup ksi k   Pole ksi k

samych priorytetach. Wiązanie lewostronne oznacza, że operacje są wykonywane w kolejności od lewej do prawej. Takim operatorem jest na przykład odejmowanie. Zatem wyrażenie:

```
w = x - y - z;
```

jest równoważne następującemu:

```
w = ((x - y) - z);
```

Natomiast wyrażenia:

```
y = a ** b ** c;
x = ~-y;
w = x = y = z;
q = a?b:c?d:e?f:g;
```

są równoważne następującym wyrażeniom:

```
y = (a ** (b ** c));
x = ~(-y);
w = (x = (y = z));
q = a?b:(c?d:(e?f:g));
```

Kolejność działań jest inna, ponieważ operatory potęgowania, przypisania oraz operatory jednoargumentowe i warunkowe trójargumentowe mają wiązania prawostronne.

4.7.6. Kolejność przetwarzania

Priorytety i wiązania operatorów określają kolejność ich stosowania w złożonych wyrażeniach, ale nie określają kolejności wyliczania wyników podwyrażeń. W języku JavaScript wyrażenia są zawsze przetwarzane od strony lewej do prawej. Na przykład w wyrażeniu `w = x + y * z` najpierw przetwarzane jest podwyrażenie `w`, a następnie `x`, `y` i `z`. W dalszej kolejności mnożone są wyrażenia `y` i `z`, do uzyskanego iloczynu jest dodawana wartość wyrażenia `x`, a uzyskany wynik jest przypisywany zmiennej lub właściwości opisanej za pomocą wyrażenia `w`. Umieszczając nawiasy w powyższym wyrażeniu, można zmienić kolejność wykonywania operacji mnożenia, dodawania i przypisania, ale nie zmienia to kolejności przetwarzania całego wyrażenia od strony lewej do prawej.

Kolejność przetwarzania ma znaczenie tylko wtedy, gdy użyte podwyrażenia wywołują efekty uboczne wpływające na wyniki innych podwyrażeń. Jeżeli na przykład wyrażenie `x` zwiększa wartość zmiennej wykorzystywanej w wyrażeniu `z`, wówczas ma znaczenie, że wyrażenie `x` jest przetwarzane przed `z`.

4.8. Operatory arytmetyczne

W tym podrozdziale opisane są operatory wykonujące na swoich operandach operacje arytmetyczne i inne działania liczbowe. Operatory potęgowania, mnożenia, dzielenia i odejmowania są proste, więc będą przedstawione w pierwszej kolejności. Operatorowi dodawania jest poświęcony osobny punkt, ponieważ służy on również do łączenia ciągów znaków i rządzą w nim nietypowe reguły konwersji typów. Operatory jednoargumentowe i bitowe będą również wyjaśnione w osobnych punktach.

Kup ksi k Pole ksi k

Większość operatorów arytmetycznych można z opisanymi niżej zastrzeżeniami stosować zarówno z operandami typu `BigInt` (patrz punkt 3.2.5), jak i zwykłymi liczbami, o ile tylko operandy nie są różnych typów.

Podstawowe operatory arytmetyczne to ** (potęgowanie), * (mnożenie), / (dzielenie), `%` (modulo, czyli reszta z dzielenia), + (dodawanie) i – (odejmowanie). Jak wspomniałem wcześniej, operator dodawania będzie opisany w osobnym punkcie. Pięć pozostałych operatorów po prostu wylicza wartości operandów, przekształca je w liczby, jeżeli jest taka potrzeba, i na koniec wylicza potęgę, iloczyn, iloraz, sumę lub różnicę. Operandy, których nie można przekształcić w liczby, są konwertowane na wartości `NaN`. Jeżeli jeden z operandów ma taką wartość (lub został w nią przekształcony), wynikiem całej operacji jest prawie zawsze `NaN`.

Operator ** ma wyższy priorytet niż operatory *, / i `%` (które z kolei mają wyższy priorytet niż + i `-`). Operator ** w odróżnieniu od pozostałych ma wiązanie prawostronne. Zatem wyrażenie `2**2**3` jest równoważne `2**8`, a nie `4**3`. Naturalna niejednoznaczność pojawia się w wyrażeniach takich jak `-3**2`. W zależności od względnych priorytetów operatorów zmiany znaku i potęgowania wyrażenie to jest równoważne `(-3)**2` lub `-(3**2)`. W różnych językach programowania przetwarzanie przebiega różnie, natomiast w języku JavaScript jest po prostu zgłaszany błąd składniowy, co zmusza programistę do użycia nawiasów i napisania jednoznacznego wyrażenia. Operator potęgowania jest jednym z najnowszych operatorów arytmetycznych — został wprowadzony w wersji ES2016. Natomiast funkcja `Math.pow()`, dostępna od najwcześniejszych wersji języka JavaScript, działa dokładne tak samo jak operator **.

Operator / dzieli pierwszy operand przez drugi. Jeżeli znasz język programowania, w którym rozróżniane są liczby całkowite i zmiennoprzecinkowe, zapewne spodziewasz się, że wynikiem dzielenia dwóch liczb całkowitych jest również liczba całkowita. Jednak w języku JavaScript wszystkie liczby są zmiennoprzecinkowe i taki też jest wynik każdej operacji dzielenia. Dlatego wynikiem wyrażenia `5/2` jest `2.5`, a nie `2`. Wynikiem dzielenia przez zero jest dodatnia lub ujemna nieskończoność, natomiast wynikiem `0/0` jest wartość `NaN`. W żadnym z tych przypadków nie jest zgłaszany wyjątek.

Operator `%` wykonuje operację modulo, czyli wylicza resztę z dzielenia pierwszego operandu przez drugi. Znak wyniku jest taki sam jak znak pierwszego operandu. Na przykład wynikiem wyrażenia `5 % 2` jest `1`, a wynikiem `-5 % 2` jest `-1`. Operator modulo stosuje się zazwyczaj z liczbami całkowitymi, ale można również używać liczb zmiennoprzecinkowych. Na przykład wynikiem wyrażenia `6.5 % 2.1` jest `0.2`.

4.8.1. Operator +

Dwuargumentowy operator + dodaje operandy liczbowe lub łączy ze sobą operandy tekstowe:

```
1 + 2                     // => 3
"Dzień" + " " + "dobry"   // => "Dzień dobry"
"1" + "2"                 // => "12"
```

Jeżeli oba operandy są liczbami lub ciągami znaków, wynik jest oczywisty. Jednak w każdym innym przypadku, kiedy wymagana jest konwersja typów, wynik operacji zależy od rodzaju konwersji. Wyższy priorytet ma konwersja na ciągi znaków. Jeżeli jeden operand jest ciągiem lub obiektem,

Kup ksi k Pole ksi k

który można przekształcić w ciąg, wówczas drugi operand jest również przekształcany w ciąg znaków i oba ciągi są ze sobą łączone. Dodawanie jest wykonywane tylko wtedy, gdy żaden operand nie jest ciągiem.

Z technicznego punktu widzenia operand + działa w następujący sposób:

- Jeżeli jeden z operandów jest obiektem, jest przekształcany w wartość prymitywną zgodnie z algorytmem opisanym w punkcie 3.9.3. Obiekt typu `Date` jest przekształcany w ciąg za pomocą jego metody `toString()`, a obiekty wszystkich pozostałych typów za pomocą metody `valueOf()`, jeżeli metoda ta zwraca wartość prymitywną. Większość obiektów nie posiada jednak przydatnej metody `valueOf()`, dlatego stosowana jest metoda `toString()`.
- Jeżeli po przekształceniu obiektów w wartości prymitywne jeden z operandów jest ciągiem znaków, to drugi operand jest również przekształcany w ciąg, a oba ciągi są ze sobą łączone.
- W przeciwnym razie oba operandy są przekształcane w liczby (lub wartość `NaN`) i wykonywane jest dodawanie.

Poniżej jest przedstawionych kilka przykładów:

```
1 + 2          // => 3: dodawanie.
"1" + "2"      // => "12": łączenie ciągów.
"1" + 2        // => "12": łączenie ciągów po przekształceniu liczby w ciąg.
1 + {}         // => "1[object Object]": łączenie ciągów po przekształceniu obiektu w ciąg.
true + true    // => 2: dodawanie po przekształceniu wartości logicznej w liczbę.
2 + null       // => 2: dodawanie po przekształceniu wartości null w 0.
2 + undefined  // => NaN: dodawanie po przekształceniu wartości undefined w NaN.
```

Pamiętaj, że w przypadku użycia liczby i ciągu znaków może nie obowiązywać wiązanie operatora. W takiej sytuacji wynik zależy od kolejności wykonania operacji, na przykład:

```
1 + 2 + " dni"     // => "3 dni"
1 + (2 + " dni")   // => "12 dni"
```

W pierwszym wyrażeniu nie ma nawiasów, więc zostało zastosowane wiązanie od lewej do prawej. Dlatego najpierw zostały dodane dwie liczby, a następnie ich suma połączona z ciągiem. W drugim wyrażeniu nawiasy zmieniły kolejność operacji: najpierw liczba 2 została połączona z ciągiem i został utworzony nowy ciąg. Potem do niego została dołączona liczba 1 i w ten sposób został wyliczony ostateczny wynik.

4.8.2. Jednoargumentowe operatory arytmetyczne

Operator jednoargumentowy modyfikuje pojedynczy operand i tworzy nową wartość. W języku JavaScript wszystkie operatory jednoargumentowe mają wysokie priorytety i prawostronne wiązania. Wszystkie tego rodzaju operatory opisane w tym punkcie (+, -, ++ i `--`) przekształcają w razie potrzeby operand w liczbę. Zwróć uwagę, że znaki + i – są zarówno operatorami jednoargumentowymi, jak i dwuargumentowymi.

Dostępne są następujące jednoargumentowe operatory arytmetyczne:

Kup ksi k Pole ksi k

Plus (+)

Jednoargumentowy operator + przekształca operand w liczbę (lub wartość `NaN`) i zwraca uzyskaną wartość. Jeżeli operand jest już liczbą, operator nic nie robi. Nie można go stosować z wartością typu `BigInt`, ponieważ nie można jej przekształcić w zwykłą liczbę.

Minus (–)

Jednoargumentowy operator – przekształca w razie potrzeby operand w liczbę, a następnie zmienia jej znak na przeciwny.

Inkrementacja (++)

Operator ++ inkrementuje, tj. zwiększa o 1, pojedynczy operand, który musi być *l*-wartością (zmienną, elementem tablicy lub właściwością obiektu). Operator przekształca operand w liczbę, dodaje do niej liczbę 1 i uzyskany wynik przypisuje z powrotem zmiennej, elementowi lub właściwości.

Wartość zwracana przez ten operator zależy od jego pozycji względem operandu. Operator umieszczony przed operandem jest operatorem preinkrementacji, tzn. najpierw zwiększa wartość operandu, a następnie zwraca uzyskaną w ten sposób wartość. Operator umieszczony za operandem jest operatorem postinkrementacji, tj. zwiększa jego wartość operandu, ale zwraca jego oryginalną wartość. Przeanalizujmy różnice pomiędzy poniższymi wierszami:

```
let i = 1, j = ++i;    // Zmienne i oraz j mają wartość 2.
let n = 1, m = n++;    // Zmienna n ma wartość 2, a zmienna m wartość 1.
```

Zwróć uwagę, że wyrażenie `x++` nie zawsze jest równoważne `x=x+1`. Operator `++` nie łączy ciągów znaków. Zawsze przekształca operand w liczbę i zwiększa jego wartość. Jeżeli `x` jest ciągiem `"1"`, to wyrażenie `++x` jest liczbą 2, natomiast `x+1` jest ciągiem `"11"`.

Pamiętaj również, że ze względu na automatyczne umieszczanie średnika w języku JavaScript, nie można wprowadzać podziału wiersza pomiędzy operator postinkrementacji a poprzedzający go operand. W takim przypadku operator zostanie potraktowany jako samodzielna instrukcja i zostanie przed nim umieszczony średnik.

Opisywany operator, zarówno pre-, jak i postinkrementacji, jest najczęściej stosowany do modyfikowania licznika w pętli `for` (patrz punkt 5.4.3).

Dekrementacja (--)

Operandem operatora — jest *l*-wartość. Operator ten przekształca operand w liczbę, odejmuje od niej liczbę 1 i uzyskany wynik przypisuje z powrotem operandowi. Podobnie jak w przypadku operatora `++`, zwracana wartość zależy od pozycji operatora względem operandu. Operator umieszczony przed operandem zmniejsza go i zwraca uzyskaną wartość. Natomiast operator umieszczony za operandem zmniejsza go, ale zwraca oryginalną wartość. Jeżeli operator znajduje się za operandem, nie można pomiędzy nimi umieszczać podziału wiersza.

4.8.3. Operatory bitowe

Operatory bitowe wykonują niskopoziomowe operacje na bitach reprezentujących liczby. Choć nie są to typowe operatory arytmetyczne, są kwalifikowane jako takie, ponieważ działają na liczbach i zwracają liczby. Cztery operatory wykonują operacje logiczne na poszczególnych bitach operandów,

Kup ksi k Pole ksi k

tj. traktują każdy bit jako osobną wartość logiczną (1 = `true`, 0 = `false`). Trzy inne operatory przesuwają bity w lewo lub prawo. Są rzadko stosowane, więc jeżeli nie znasz zasad binarnego wyrażania liczb całkowitych, w tym liczb ujemnych, możesz pominąć ten punkt.

Operandami operatora bitowego są 32-bitowe liczby całkowite, a nie 64-bitowe liczby zmiennoprzecinkowe. Operator przekształca w razie potrzeby operandy w liczby, a następnie zaokrągla je do 32-bitowych liczb całkowitych poprzez odrzucenie części ułamkowych lub bitów na pozycjach 32 i wyższych. Operatory przesunięcia wymagają, aby operand po prawej stronie był liczbą z zakresu od 0 do 31. Po przekształceniu go w 32-bitową liczbę całkowitą odrzucane są bity na pozycjach wyższych niż piąta, dzięki czemu uzyskiwana jest wartość należąca do wymaganego zakresu. Co ciekawe, wartości `NaN`, `Infinity` i `-Infinity` są przekształcane w liczbę zero.

Wszystkie operatory z wyjątkiem >>> można stosować ze zwykłymi liczbami lub operandami typu `BigInt` (patrz punkt 3.2.5).

Bitowa operacja ORAZ (&)

Operator `&` wykonuje logiczną operację ORAZ na poszczególnych bitach operandów. Ustawia bit wyniku na 1 tylko wtedy, gdy oba odpowiadające sobie bity operandów są równe 1. Na przykład wynikiem wyrażenia `0x1234 & 0x00FF` jest `0x0034`.

Bitowa operacja LUB (|)

Operator `|` wykonuje logiczną operację LUB na poszczególnych bitach operandów. Ustawia bit wyniku na 1 tylko wtedy, gdy przynajmniej jeden z odpowiadających sobie bitów operandów jest równy 1. Na przykład wynikiem wyrażenia `0x1234 | 0x00FF` jest `0x12FF`.

Bitowa różnica symetryczna (^)

Operator `^` wylicza różnicę symetryczną poszczególnych bitów operandów. Różnica symetryczna ma miejsce wtedy, gdy tylko jeden z operandów ma wartość `true`. Operator ustawia bit wyniku na 1 tylko wtedy, gdy dokładnie jeden z odpowiadających sobie bitów operandów jest równy 1. Na przykład wynikiem wyrażenia `0xFF00 ^ 0xF0F0` jest `0x0FF0`.

Bitowa operacja NIE (~)

Operator `~` jest operatorem jednoargumentowym, umieszczanym przed operandem. Odwraca wszystkie bity operandu. Ze względu na sposób wyrażania w języku JavaScript znaku liczby całkowitej, użycie operatora `~` jest równoważne zmianie znaku operandu i odjęciu od niego liczby 1. Na przykład wynikiem wyrażenia `~0x0F` jest liczba `0xFFFFFFF0`, czyli `−16`.

Przesunięcie bitów w lewo (<<)

Operator `<<` przesuwa wszystkie bity lewego operandu w lewo o zadaną liczbą pozycji, określoną za pomocą prawego operandu, która powinna zawierać się w przedziale od 0 do 31. Na przykład wyrażenie `a << 1` powoduje, że pierwszy bit (licząc od prawej) jest przesuwany na drugą pozycję, drugi na trzecią itd. Na pierwszej pozycji umieszczane jest zero, a bit z 32. pozycji jest odrzucany. Przesunięcie o jeden bit jest równoważne mnożeniu przez 2, o dwa bity mnożeniu przez 4 itd. Na przykład wynikiem wyrażenia `7 << 2` jest liczba `28`.

Kup ksi k Pole ksi k

Przesunięcie bitów w prawo z zachowaniem znaku (>>)

Operator >> przesuwa wszystkie bity lewego operandu w prawo o zadaną liczbę pozycji, określoną za pomocą prawego operandu, która powinna zawierać się w przedziale od 0 do 31. Skrajne prawe bity są odrzucane, natomiast wartości bitów wstawianych z lewej strony zależą od początkowego znaku operandu. Celem jest uzyskanie prawidłowego znaku wyniku. Jeżeli lewy operand jest dodatni, wstawiane są zera, a w przeciwnym razie jedynki. Przesunięcie dodatniej wartości w prawo o jeden bit jest równoważne podzieleniu jej przez 2 i odrzuceniu reszty, przesunięcie o dwa bity — równoważne podzieleniu przez 4 itd. Wynikiem wyrażenia `7 >> 1` jest liczba `3`. Zwróć jednak uwagę, że wynikiem wyrażenia `−7 >> 1` jest liczba `−4`.

Przesunięcie bitów w prawo z uzupełnieniem zerami (>>>)

Operator >>> działa podobnie jak >> z tą różnicą, że bity po lewej stronie uzupełnia zerami, niezależnie od znaku operandu. Jest to ważne w sytuacji, gdy 32-bitowa liczba całkowita powinna być traktowana jako liczba bez znaku. Na przykład wynikiem wyrażenia `−1 >> 4` jest liczba –1, ale wynikiem `−1 >>> 4` jest `0x0FFFFFFF`. Jest to jedyny operator bitowy, którego nie można użyć z operandem typu `BigInt`, ponieważ w tym typie liczba ujemna nie ma ustawionego najbardziej znaczącego bitu na 1, jak w przypadku zwykłej 32-bitowej liczby całkowitej. Stosowanie tego operatora ma sens jedynie wtedy, gdy liczba jest kodowana w systemie uzupełnienia do dwóch.

4.9. Wyrażenia relacyjne

W tym podrozdziale opisane są operatory relacyjne wykorzystywane do sprawdzania zależności pomiędzy dwiema wartościami (na przykład „równe”, „mniejsze niż”, „właściwość obiektu”). Zwracanym wynikiem jest `true` lub `false` w zależności od relacji pomiędzy wartościami. Wynikiem wyrażenia relacyjnego jest zawsze wartość logiczna. Tego rodzaju wyrażenia są często wykorzystywane w instrukcjach `if`, `while` i `for` sterujących przepływem programu (patrz rozdział 5.). Wkolejnych punktach są opisane operatory równości, nierówności, porównania oraz `in` i `instanceof`.

4.9.1. Operatory równości i nierówności

Operatory == i === sprawdzają, czy dwie wartości są sobie równe, ale każdy stosuje inną definicję równości. Operandy mogą być dowolnych typów, a oba operatory zwracają wartość `true`, jeżeli operandy są sobie równe, lub `false` w przeciwnym razie. Operator === jest nazywany operatorem **ścisłej równości** (lub **tożsamości**), ponieważ sprawdza identyczność operandów, wykorzystując ścisłą definicję równości. Operator == jest nazywany po prostu operatorem **równości**. Sprawdza, czy dwa operandy są sobie równe, wykorzystując mniej restrykcyjną definicję równości, dopuszczającą konwersję typów.

Operatory `!=` i `!==` funkcjonują odwrotnie do operatorów `==` i `===`. Operator nierówności `!=` zwraca wartość `false`, gdy oba operandy są według operatora == sobie równe. W przeciwnym razie zwraca wartość `true`. Operator `!==` zwraca wartość `false`, gdy oba operandy są ściśle sobie równe. W przeciwnym razie zwraca `true`. Jak się przekonasz w podrozdziale 4.10, operator `!`

Kup ksi k Pole ksi k

wykonuje logiczną operację NIE. Łatwo więc zapamiętać, że operatory `!=` i `!==` oznaczają, odpowiednio, „nie jest równe” i „nie jest ściśle równe”.

> **Operatory =, == i ===**
>
> W języku JavaScript można stosować operatory `=`, `==` i `===`. Ważna jest świadomość różnic pomiędzy przypisaniem, równością i ścisłą równością, aby podczas kodowania używać właściwych operatorów. Choć wszystkie trzy operatory zazwyczaj czyta się jako „jest równe”, warto w celu uniknięcia nieporozumień stosować dla operatora `=` określenie „uzyskuje” lub „przypisuje”, dla `==` — określenie „jest równe”, a dla `===` — „jest ściśle równe”.
>
> Operator `==` jest przestarzałą funkcjonalnością języka JavaScript, powszechnie uważaną za źródło błędów. Niemal zawsze należy zamiast niego stosować operator `===`, a zamiast `!=` operator `!==`.

Jak wspomniałem w podrozdziale 3.8, obiekty w języku JavaScript są porównywane na podstawie referencji, a nie wartości. Obiekt jest równy samemu sobie, ale nie innemu obiektowi. Nawet obiekty posiadające tyle samo właściwości o takich samych nazwach nie są sobie równe. Analogicznie dwie tablice zawierające takie same elementy ułożone w tej samej kolejności nie są sobie równe.

Ścisła równość

Operator ścisłej równości `===` wylicza wartości operandów, a następnie bez przekształcania typów sprawdza ich równość w następujący sposób:

- Jeżeli wartości są różnych typów, są uznawane za różne.
- Jeżeli obie wartości są równe `null` lub obie są równe `undefined`, są uznawane za równe sobie.
- Jeżeli dwie logiczne wartości są równe `true` lub obie są równe `false`, są uznawane za równe sobie.
- Jeżeli przynajmniej jedna wartość jest równa `NaN`, obie są uznawane za różne. (Może się to wydawać dziwne, ale wartość `NaN` nie jest równa żadnej innej wartości, również `NaN`. Aby sprawdzić, czy zmienna `x` ma wartość `NaN`, należy użyć wyrażenia `x !== x` lub globalnej funkcji `isNaN()`).
- Jeżeli obie wartości są takimi samymi liczbami, są uznawane za równe sobie. Jeżeli jedna z nich jest równa 0, a druga –0, również są uznawane za równe sobie.
- Jeżeli obie wartości są ciągami zawierającymi dokładnie takie same 16-bitowe wartości (patrz ramka w podrozdziale 3.3) na tych samych pozycjach, są uznawane za równe sobie. Jednak dwa ciągi zawierające tę samą treść i wyglądające tak samo mogą być zakodowane na różne sposoby. W języku JavaScript nie jest przeprowadzana normalizacja ciągów i takie ciągi dla operatorów `==` i `===` są różne.
- Jeżeli obie wartości odwołują się do tego samego obiektu, tablicy lub funkcji, są uznawane za równe sobie. Jeżeli odwołują się do różnych obiektów posiadających identyczne właściwości, są uznawane za różne.

Kup ksi k Pole ksi k

Równość z konwersją typów

Operator == jest mniej ścisły od operatora ===. Jeżeli operandy są różnych typów, operator dokonuje konwersji i ponownie porównuje wartości:

- Jeżeli obie wartości są tego samego typu, sprawdzana jest ich równość w opisany wyżej sposób.
- Jeżeli wartości są różnych typów, stosowane są następujące reguły sprawdzania równości:
  - Jeżeli jedna wartość jest równa `null`, a druga `undefined`, obie są uznawane za równe sobie.
  - Jeżeli jedna wartość jest liczbą, a druga ciągiem znaków, ciąg jest przekształcany w liczbę, a następnie obie wartości są porównywane ponownie.
  - Jeżeli któraś z wartości jest równa `true`, jest przekształcana w liczbę 1, a następnie obie wartości są porównywane ponownie. Jeżeli któraś z wartości jest równa `false`, jest przekształcana w liczbę 0, a następnie obie wartości są porównywane ponownie.
  - Jeżeli jedna wartość jest obiektem, a druga liczbą lub ciągiem znaków, obiekt jest przekształcany w wartość prymitywną zgodnie z algorytmem opisanym w punkcie 3.9.3, a następnie obie wartości są porównywane ponownie. Obiekt jest przekształcany w wartość prymitywną poprzez wywołanie jego metody `toString()` lub `valueOf()`. W przypadku wbudowanych klas najpierw jest wywoływana metoda `valueOf()`. Wyjątkiem jest klasa `Date`, w przypadku której najpierw jest wywoływana metoda `toString()`.
  - Wszystkie inne kombinacje wartości są uznawane za różne.

Przeanalizujmy dla przykładu następujące porównanie:

```
"1" == true  // => true
```

Wynikiem powyższego wyrażenia jest wartość `true` oznaczająca, że te dwie, tak odmiennie wyglądające wartości są sobie równe. Najpierw wartość logiczna `true` jest przekształcana w liczbę 1 i ponownie porównywana. Następnie ciąg znaków `"1"` jest przekształcany w liczbę 1. Ponieważ obie uzyskane w ten sposób wartości są takie same, wynikiem całego wyrażenia jest wartość `true`.

4.9.2. Operatory porównania

Operatory porównania sprawdzają kolejność (liczbową lub tekstową) dwóch operandów:

Mniejszy niż (<)

Operator < zwraca wartość `true`, jeżeli lewy operator jest mniejszy od prawego. W przeciwnym razie zwraca wartość `false`.

Większy niż (>)

Operator > zwraca wartość `true`, jeżeli lewy operator jest większy od prawego. W przeciwnym razie zwraca wartość `false`.

Mniejszy lub równy (<=)

Operator <= zwraca wartość `true`, jeżeli lewy operator jest mniejszy lub równy prawemu. W przeciwnym razie zwraca wartość `false`.

Kup ksi k Pole ksi k

Większy lub równy niż (>=)

Operator `>=` zwraca wartość `true`, jeżeli lewy operator jest większy lub równy prawemu. W przeciwnym razie zwraca wartość `false`.

Powyższe operatory działają na operandach dowolnych typów. Porównywane mogą być tylko liczby i ciągi znaków, więc wszystkie inne typy są przekształcane.

Porównywanie i przekształcanie operandów przebiega następująco:

- Jeżeli któryś z operandów jest obiektem, jest przekształcany w wartość prymitywną w sposób opisany na końcu punktu 3.9.3. Jeżeli jego metoda `valueOf()` zwraca wartość prymitywną, jest ona wykorzystywana w porównaniu. W przeciwnym razie wykorzystywana jest wartość zwrócona przez metodę `toString()`.
- Jeżeli po dokonaniu powyższego przekształcenia oba operandy są ciągami znaków, są ze sobą porównywane. Stosowany jest przy tym porządek alfabetyczny określony za pomocą 16-bitowych wartości Unicode, z których składają się ciągi.
- Jeżeli po dokonaniu powyższego przekształcenia oba operandy nie są ciągami znaków, są przekształcane w liczby i porównywane ponownie. Wartości `0` i `-0` są uznawane za sobie równe. Wartości `Infinity` i `-Infinity` są uznawane, odpowiednio, za większą i mniejszą niż dowolna liczba. Jeżeli przynajmniej jeden z operandów ma wartość `NaN` (lub został w nią przekształcony), operator zwraca wartość `false`. W odróżnieniu od operatorów arytmetycznych można mieszać operandy typu `BigInt` i zwykłe liczby.

Pamiętaj, że ciągi znaków w języku JavaScript są sekwencjami 16-bitowych wartości, więc porównywanie ciągów polega w rzeczywistości na porównywaniu wartości liczbowych. Kodowanie zdefiniowane w standardzie Unicode może różnić się od kodowania dla danego języka lub ustawień regionalnych. Zwróć przede wszystkim uwagę, że przy porównywaniu uwzględniana jest wielkość liter. Wszystkie wielkie litery w zestawie ASCII są „mniejsze” od małych liter. Nieuwzględnienie tej zasady może skutkować uzyskaniem niespodziewanych wyników. Na przykład dla operatora `<` ciąg `"Zoo"` jest mniejszy niż `"aardvark"`.

Jeżeli potrzebny jest bardziej uniwersalny algorytm porównywania ciągów, należy użyć metody `String.localeCompare()` uwzględniającej kolejność alfabetyczną właściwą dla danego języka. Aby porównać ciągi bez względu na wielkości liter, należy je przekształcić w wielkie lub małe litery, odpowiednio, za pomocą metod `String.toLowerCase()` lub `String.toUpperCase()`. Jeszcze lepszym narzędziem do porównywania ciągów charakterystycznych dla danego języka jest klasa `Intl.Collator`, która będzie opisana w punkcie 11.7.3.

Operator `+` i operatory porównania funkcjonują odmiennie w przypadku operandów liczbowych i tekstowych. Operator `+` faworyzuje ciągi znaków, tj. łączy je, jeżeli przynajmniej jeden z operandów jest ciągiem. Operatory porównania faworyzują liczby i działają na ciągach tylko wtedy, gdy oba operandy są ciągami:

```
1 + 2        // => 3: dodawanie.
"1" + "2"    // => "12": łączenie ciągów.
"1" + 2      // => "12": liczba 2 jest przekształcana w ciąg "2".
11 < 3       // => false: porównywanie liczb.
"11" < "3"   // => true: porównywanie ciągów.
"11" < 3     // => false: porównywanie liczb, ciąg "11" jest przekształcany w liczbę 11.
"jeden" < 3  // => false: porównywanie liczb, ciąg "jeden" jest przekształcany w wartość NaN.
```


Kup ksi k Pole ksi k

Na koniec zwróć uwagę, że operatory <= (mniejszy lub równy) i >= (większy lub równy) nie stosują reguł równości ani ścisłej równości do określenia, czy obie wartości są „równe”. Pierwszy z operatorów jest po prostu zdefiniowany jako „nie większy niż”, a drugi „nie mniejszy niż”. Wyjątkiem jest sytuacja, w której jeden lub oba operandy mają wartość `NaN` (lub są w nią przekształcane). Wtedy każdy z czterech operatorów zwraca wartość `false`.

4.9.3. Operator in

Lewym operandem operatora `in` powinien być ciąg znaków, symbol lub wartość, którą można przekształcić w ciąg. Natomiast prawym operandem powinien być obiekt. Operator zwraca wartość `true`, jeżeli wartością znajdującą się po lewej stronie jest nazwa właściwości obiektu znajdującego się po stronie prawej, na przykład:

```
let point = {x: 1, y: 1};  // Definicja obiektu.
"x" in point               // => true: obiekt ma właściwość o nazwie "x".
"z" in point               // => false: obiekt nie ma właściwości o nazwie "z".
"toString" in point        // => true: obiekt dziedziczy metodę toString().
let data = [7,8,9];        // Tablica elementów o indeksach 0, 1 i 2.
"0" in data                // => true: tablica zawiera element o indeksie "0".
1 in data                  // => true: liczby są przekształcane w ciągi znaków.
3 in data                  // => false: tablica nie zawiera elementu o indeksie 3.
```

4.9.4. Operator instanceof

Operand umieszczony po lewej stronie operatora `instanceof` powinien być obiektem, a operand po prawej — klasą. Operator zwraca wartość `true`, jeżeli operand po lewej stronie jest instancją klasy po stronie prawej. W przeciwnym razie zwraca wartość `false`. W rozdziale 9. wyjaśnię, jak definiuje się klasy i inicjuje je za pomocą konstruktorów. Jak się przekonasz, operand umieszczony po prawej stronie powinien być funkcją. Poniżej znajduje się kilka przykładów:

```
let d = new Date();  // Utworzenie nowego obiektu za pomocą konstruktora Date().
d instanceof Date    // => true: obiekt d został utworzony za pomocą konstruktora Date().
d instanceof Object  // => true: wszystkie obiekty są instancjami klasy Object.
d instanceof Number  // => false: obiekt d nie jest instancją klasy Number.
let a = [1, 2, 3];   // Utworzenie tablicy za pomocą literału.
a instanceof Array   // => true: a jest tablicą.
a instanceof Object  // => true: wszystkie tablice są obiektami.
a instanceof RegExp  // => false: tablice nie są wyrażeniami regularnymi.
```

Zwróć uwagę, że wszystkie obiekty są instancjami klasy `Object`. Operator `instanceof` uwzględnia klasy nadrzędne. Jeżeli operand po lewej stronie operatora nie jest obiektem, operator zwraca wartość `false`. Jeżeli operand po prawej stronie nie jest klasą, zgłaszany jest wyjątek TypeError.

Aby zrozumieć zasadę działania operatora `instanceof`, trzeba najpierw poznać pojęcie **łańcucha prototypów**. Jest to mechanizm dziedziczenia klas, który będzie opisany w punkcie 6.3.2. Interpreter języka JavaScript, wyliczając wynik wyrażenia `o instanceof f`, przetwarza wyrażenie `f.prototype` i sprawdza, czy uzyskany wynik znajduje się w łańcuchu prototypów obiektu `o`. Jeżeli tak, operator uznaje, że obiekt `o` jest instancją klasy `f` (lub jej podklasy) i zwraca wartość `true`. W przeciwnym razie zwraca wartość `false`.

Kup ksi k   Pole ksi k

4.10. Wyrażenia logiczne

Operatory `&&`, `||` i `!` wykonują operacje logiczne i są często stosowane łącznie z operatorami relacyjnymi. W ten sposób dwa wyrażenia relacyjne można połączyć w jedno złożone wyrażenie. Operatory logiczne są opisane w kolejnych punktach. Jednak aby je w pełni zrozumieć, przypomnij sobie wprowadzone w podrozdziale 3.4 pojęcia wartości „prawdziwych” i „fałszywych”.

4.10.1. Operator logiczny ORAZ (&&)

Operator `&&` można rozpatrywać w trzech różnych kontekstach. W najprostszym, gdy operandami są wartości logiczne, operator ten wykonuje logiczną operację ORAZ i zwraca wartość `true` tylko wtedy, gdy oba operandy reprezentują wartości `true`. Jeżeli przynajmniej jeden z nich ma wartość `false`, operator również zwraca wartość `false`.

Operator `&&` jest często stosowany do łączenia dwóch wyrażeń relacyjnych, na przykład:

```
x === 0 && y === 0    // true tylko wtedy, gdy zarówno x, jak i y są równe 0.
```

Wynikiem wyrażenia relacyjnego jest zawsze wartość `true` lub `false`. Dlatego w wyrażeniu takim jak powyższe operator `&&` zwraca jedną z tych wartości. Operator relacyjny ma wyższy priorytet od operatora `&&` (i `||`), więc wyrażenia takie jak powyższe można bez obaw wpisywać bez nawiasów.

Jednak operator `&&` nie wymaga, aby oba operandy były wartościami logicznymi. Jak już wiesz, wszystkie wartości w języku JavaScript są „prawdziwe” lub „fałszywe”. (Szczegółowe informacje na ten temat były przedstawione w podrozdziale 3.4. Wartości „fałszywe” to `false`, `null`, `undefined`, `0`, `-0`, `NaN` i `""`. Wszystkie pozostałe, włącznie z obiektami, to wartości „prawdziwe”). Drugi kontekst, w którym można interpretować operator `&&`, to wykonywanie logicznej operacji ORAZ na wartościach prawdziwych i fałszywych. Jeżeli oba operandy są prawdziwe, operator zwraca prawdziwy wynik. Jeżeli przynajmniej jeden z operandów jest fałszywy, operator zwraca fałszywy wynik. Każde wyrażenie lub instrukcja operująca na wartościach logicznych operuje również na wartościach prawdziwych i fałszywych. Zatem operator `&&`, który nie zawsze zwraca wartość `true` lub `false`, w praktyce nie powoduje problemów.

Zwróć uwagę, że w powyższym opisie zostały użyte zwroty „operator zwraca wartość prawdziwą” lub „fałszywą” bez uszczegółowienia, co te pojęcia oznaczają. Dlatego operator `&&` można rozpatrywać w jeszcze jednym, najbardziej ogólnym kontekście. Operator najpierw wylicza wartość lewego operandu. Jeżeli jest fałszywa, oznacza to, że wartość całego wyrażenia jest również fałszywa, więc operator zwraca po prostu wartość lewego operandu bez wyliczania wartości prawego operandu.

Jeżeli natomiast wartość lewego operandu jest prawdziwa, wówczas wartość całego wyrażenia zależy od wartości prawego operandu. Jeżeli jest prawdziwa, wartość całego wyrażenia jest również prawdziwa. W przeciwnym razie wartość wyrażenia jest fałszywa. Zatem gdy wartość po lewej stronie jest prawdziwa, operator `&&` wylicza i zwraca wartość znajdującą się po jego prawej stronie:

```
let o = {x: 1};
let p = null;
o && o.x      // => 1: o ma wartość prawdziwą, więc operator zwraca wartość o.x.
p && p.x      // => null: p ma wartość fałszywą, więc operator nie wylicza wartości p.x.
```


Kup ksi k Pole ksi k

Ważne jest, że operator && może, ale nie musi wyliczać wartości operandu znajdującego się po jego prawej stronie. W powyższym przykładzie zmiennej p została przypisana wartość null, więc próba wyliczenia wartości wyrażenia p.x spowodowałaby zgłoszenie wyjątku TypeError. Jednak operator jest stosowany w sposób idiomatyczny, przez co wyrażenie p.x jest wyliczane tylko wtedy, gdy zmienna p ma wartość prawdziwą, czyli inną niż null i undefined.

Działanie operatora && jest czasami nazywane „krótkim zwarciem". Możesz mieć do czynienia z kodem, w którym ta cecha operatora będzie świadomie wykorzystana do warunkowego wykonywania kodu. Na przykład poniższe dwa wiersze dają ten sam efekt:

```
if (a === b) stop();    // Funkcja stop() jest wywoływana tylko wtedy, gdy a === b.
(a === b) && stop();    // Ta instrukcja działa tak samo jak powyższa.
```

Ogólnie jednak należy zachowywać ostrożność, tworząc wyrażenia z operatorem && i operandem po prawej stronie wywołującym skutki uboczne (przypisanie, inkrementację, dekrementację lub wywołanie funkcji), ponieważ od operandu po lewej stronie będzie zależało, czy te skutki będą miały miejsce.

Operator &&, mimo dość złożonego mechanizmu działania, jest najczęściej wykorzystywany do wykonywania prostych operacji logicznych na wartościach prawdziwych i fałszywych.

4.10.2. Operator logiczny LUB (||)

Operator || wykonuje operację logiczną LUB na dwóch operandach. Jeżeli przynajmniej jeden z nich ma wartość prawdziwą, operator również zwraca wartość prawdziwą. Jeżeli oba operandy mają wartości fałszywe, operator zwraca wartość fałszywą.

Choć operator || jest najczęściej wykorzystywany jako logiczny operator LUB, jego mechanizm działania, podobnie jak operatora &&, jest dość skomplikowany. Operator najpierw wylicza wartość lewego operandu. Jeżeli jest prawdziwa, robi „krótkie zwarcie", tj. zwraca wartość prawdziwą bez wyliczania wartości prawego operandu. Jeżeli natomiast wartość lewego operandu jest fałszywa, operator wylicza i zwraca wartość prawego operandu.

Podobnie jak w przypadku operatora &&, należy unikać stosowania po prawej stronie operandu wywołującego efekty uboczne, chyba że celowo wykorzystywany jest fakt, że nie zawsze wartość takiego operandu jest wyliczana.

Idiomatyczne użycie operatora || polega na wybraniu pierwszej prawdziwej wartości w całym zestawie:

```
// Jeżeli zmienna maxWidth jest prawdziwa, użyj jej. W przeciwnym razie poszukaj jej
// w obiekcie preferences. Jeżeli nie jest prawdziwa, użyj wpisanej stałej.
let max = maxWidth || preferences.maxWidth || 500;
```

Zwróć uwagę, że jeżeli zero jest poprawną wartością zmiennej maxWidth, to kod nie będzie działał właściwie, ponieważ 0 jest wartością fałszywą. Alternatywnym rozwiązaniem jest użycie operatora ?? (patrz punkt 4.13.2).

W wersjach języka starszych niż ES6 powyższy zapis był często stosowany w funkcjach do nadawania parametrom domyślnych wartości:

Kup ksi k Pole ksi k

```
// Kopiowanie właściwości obiektu o do p i zwrócenie p.
function copy(o, p) {
  p = p || {};  // Jeżeli parametr p jest pusty, użyj nowo utworzonego obiektu.
  // Ciało funkcji.
}
```

Począwszy od wersji ES6 nie trzeba stosować powyższych sztuczek, ponieważ domyślnie wartości parametrów można wprost określić w definicji funkcji, na przykład: `function copy(o, p={}) { ... }`.

4.10.3. Operator logiczny NIE (!)

Operator `!` jest operatorem jednoargumentowym umieszczanym przed operandem. Jego zadaniem jest negowanie wartości logicznej operandu. Jeżeli na przykład zmienna `x` ma wartość prawdziwą, wyrażenie `!x` ma wartość `false`. Jeżeli natomiast `x` ma wartość fałszywą, wyrażenie `!x` ma wartość `true`.

Operator `!`, w odróżnieniu od operatorów `&&` i `||`, przekształca operand w wartość logiczną (zgodnie z regułami opisanymi w rozdziale 3.), a następnie ją neguje. Oznacza to, że operator zwraca wartość `true` lub `false`, a operand można przekształcić w równoważną mu wartość logiczną, stosując operator dwukrotnie: `!xx` (patrz punkt 3.9.2).

Operator `!` ma wysoki priorytet i ściśle wiąże operand. Aby na przykład zanegować wartość wyrażenia `p && q`, trzeba użyć nawiasów: `!(p && q)`. Zauważ, że stosując składnię języka JavaScript, można wyrazić dwa prawa algebry Boole'a:

```
// Prawa De Morgana.
!(p && q) === (!p || !q)  // => true: dla wszystkich wartości p i q.
!(p || q) === (!p && !q)  // => true: dla wszystkich wartości p i q.
```

4.11. Wyrażenia przypisujące

W języku JavaScript do przypisywania wartości zmiennej lub właściwości służy operator `=`, na przykład:

```
i = 0;     // Przypisanie zmiennej i wartości 0.
o.x = 1;   // Przypisanie właściwości x obiektu o wartości 1.
```

Operator `=` wymaga, aby operand po jego lewej stronie był *l*-wartością, tzn. zmienną, właściwością obiektu lub elementem tablicy. Wynikiem operatora przypisania jest wartość operandu znajdującego się po jego prawej stronie. Efektem ubocznym wywoływanym przez operator `=` jest przypisanie zmiennej lub właściwości umieszczonej po jego lewej stronie wartości, która jest umieszczona po jego prawej stronie. Zatem przyszłe odwołania do zmiennej lub właściwości będą zwracały nową wartość.

Choć wyrażenia przypisujące są zazwyczaj bardzo proste, czasami ich wartości są wykorzystywane w bardziej złożonych wyrażeniach. Na przykład w jednym wierszu można przypisać wartość zmiennej `a` i sprawdzić ją:

```
(a = b) === 0
```


Kup ksi k Pole ksi k

Stosując tego rodzaju zapisy, należy być świadomym różnic pomiędzy operatorami = i ===. Zwróć uwagę, że operator = ma bardzo niski priorytet. Jeżeli wynik przypisania ma być wykorzystany w większym wyrażeniu, niezbędne jest użycie nawiasów.

Operator przypisania ma wiązanie prawostronne, co oznacza, że w przypadku użycia kilku operatorów przypisania w jednym wyrażeniu, wartości są wyliczane w kolejności od prawej do lewej. Zatem w celu przypisania tej samej wartości kilku zmiennym można użyć następującego kodu:

```
i = j = k = 0;          // Zainicjowanie trzech zmiennych liczbą 0.
```

4.11.1. Przypisanie z działaniem

W języku JavaScript oprócz zwykłego operatora przypisania = dostępnych jest kilka dodatkowych operatorów łączących przypisanie z innym działaniem. Na przykład operator += wykonuje dodawanie i przypisanie. Zatem poniższe wyrażenie:

```
total += salesTax;
```

jest równoważne następującemu:

```
total = total + salesTax;
```

Jak się można spodziewać, operator += działa na liczbach i ciągach znaków. W przypadku operandów liczbowych dodaje i przypisuje wartości, a w przypadku ciągów łączy je i przypisuje.

Podobne operatory to -=, *=, &= i kilka innych. Wszystkie są wymienione w tabeli 4.2.

Tabela 4.2. Operatory przypisania z działaniem

Operator	Przykład	Odpowiednik
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
**=	a **= b	a = a ** b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

W większości przypadków wyrażenie:

```
a op= b
```

gdzie op oznacza operator, jest równoważne wyrażeniu:

```
a = a op b
```


Kup książkę Poleć książkę

W pierwszym przypadku wyrażenie a jest wyliczane jeden raz, natomiast w drugim dwa razy. Oba przypadki różnią się tylko wtedy, gdy wyrażenie a wywołuje efekty uboczne, na przykład jest to funkcja wykorzystująca operator inkrementacji. Na przykład poniższe dwa wyrażenia przypisujące nie są sobie równoważne:

```
data[i++] *= 2;
data[i++] = data[i++] * 2;
```

4.12. Wyrażenia interpretujące

W języku JavaScript, podobnie jak w wielu innych językach interpretowanych, można w ciągach znaków umieszczać kod źródłowy, uruchamiać go i odczytywać zwracane przez niego wyniki. Służy do tego celu globalna funkcja `eval()`:

```
eval("3+2")     // => 5
```

Dynamiczne interpretowanie ciągów zawierających kod źródłowy to potężna funkcjonalność, jednak w praktyce niemal zbędna. Zanim użyjesz funkcji `eval()`, dokładnie się zastanów, czy rzeczywiście jest Ci ona potrzebna. Funkcja ta stanowi potencjalną lukę w bezpieczeństwie kodu, dlatego nie można w jej argumencie umieszczać ciągów wprowadzanych przez użytkownika. W przypadku języka tak skompilowanego jak JavaScript nie ma możliwości oczyszczania uzyskiwanych w opisany sposób danych i bezpiecznego ich stosowania z funkcją `eval()`. Ze względów bezpieczeństwa niektóre serwery WWW stosują nagłówek HTTP „Content-Security-Policy” blokujący użycie funkcji `eval()` w całej witrynie.

W kolejnych punktach opisane są podstawy korzystania z funkcji `eval()` wraz z jej dwiema ograniczonymi wersjami, które w mniejszym stopniu wpływają na optymalizator języka JavaScript.

Czy funkcja eval() jest operatorem?

Funkcja `eval()` znalazła się w niniejszym rozdziale poświęconym wyrażeniom, ponieważ tak naprawdę powinna być operatorem. Została wprowadzona w jednej z pierwszych wersji JavaScriptu i od tamtej pory twórcy języka i interpretera nakładają na nią kolejne ograniczenia sprawiające, że coraz bardziej przypomina ona operator. Nowoczesne interpretery przeprowadzają bardzo zaawansowaną analizę i optymalizację kodu źródłowego. Jednak interpreter nie może zoptymalizować funkcji wywołującej funkcję `eval()`. Problem z nią polega na tym, że można jej nadać inną nazwę:

```
let f = eval;
let g = f;
```

W takim wypadku interpreter „nie jest pewien”, która funkcja wywołuje funkcję `eval()`, i nie może jej agresywnie zoptymalizować. Problemu można byłoby uniknąć, gdyby funkcja `eval()` była operatorem lub zarezerwowanym słowem. W punktach 4.12.2 i 4.12.3 poznasz ograniczenia nałożone na tę funkcję sprawiające, że bardziej przypomina ona operator.

Kup ksi k Pole ksi k

4.12.1. Funkcja eval()

Funkcja `eval()` ma jeden argument. Jeżeli umieści się w nim wartość inną niż ciąg znaków, funkcja po prostu zwróci tę wartość. Jeżeli natomiast będzie to ciąg, funkcja podejmie próbę zinterpretowania go jako kodu źródłowego JavaScript i w przypadku niepowodzenia zgłosi wyjątek SyntaxError. W przeciwnym razie uruchomi kod i zwróci wynik ostatniej zawartej w nim instrukcji lub wyrażenia albo wartość `undefined`, jeżeli takiego wyniku nie będzie. Jeżeli wykonywany kod zgłosi wyjątek, zostanie on przejęty i zgłoszony przez funkcję `eval()`.

Funkcja `eval()` ma tę cechę, że wykorzystuje zmienne kodu, który ją wywołuje. Oznacza to, że może odczytywać wartości tych zmiennych, jak również definiować nowe zmienne i funkcje w taki sam sposób, jak to się dzieje w lokalnym kodzie. Jeżeli w funkcji jest zdefiniowana lokalna zmienna `x`, wynikiem zwróconym przez funkcję `eval("x")` będzie wartość tej zmiennej. Jeżeli zostanie wywołana funkcja `eval("x=1")`, zmieni się wartość lokalnej zmiennej. Natomiast wywołanie `eval("var y = 3;")` spowoduje zadeklarowanie nowej zmiennej lokalnej `y`. Z drugiej strony, jeżeli w ciągu zostanie użyta instrukcja `let` lub `const`, zostanie zadeklarowana lokalna w kontekście funkcji zmienna lub stała, która nie będzie dostępna w wywołującym ją kodzie.

Analogicznie funkcja `eval()` może definiować własne lokalne funkcje, jak niżej:

```
eval("function f() { return x+1; }");
```

Oczywiście funkcja `eval()` wywołana na najwyższy poziomie kodu operuje na globalnych zmiennych i funkcjach.

Zwróć uwagę, że kod umieszczany w argumencie funkcji `eval()` musi stanowić syntaktyczną całość. Nie można w nim umieszczać fragmentów kodu. Na przykład zapis `eval("return;")` nie ma sensu, ponieważ instrukcję `return` stosuje się wyłącznie wewnątrz funkcji. Fakt, że kod umieszczony w argumencie funkcji `eval()` wykorzystuje to samo środowisko co wywołująca ją funkcja, nie oznacza, że kod w argumencie jest częścią tej funkcji. Jeżeli ciąg może być samodzielnym skryptem, nawet najkrótszym, na przykład `x=0`, można go umieścić w argumencie funkcji `eval()`. W przeciwnym razie zostanie zgłoszony wyjątek SyntaxError.

4.12.2. Globalne wywołanie funkcji eval()

To właśnie możliwość modyfikowania przez funkcję `eval()` lokalnych zmiennych jest tak problematyczna dla optymalizatora kodu JavaScript. Dlatego funkcje wywołujące tę funkcję są w mniejszym stopniu optymalizowane. Co może jednak zrobić interpreter, jeżeli w kodzie jest zdefiniowany alias funkcji `eval()` i użyty w celu jej wywołania? Zgodnie ze specyfikacją języka JavaScript funkcja `eval()` wywołana przy użyciu innej nazwy niż `eval` powinna traktować umieszczony w jej argumencie ciąg, tak jakby był uruchamiany na najwyższym, globalnym poziomie kodu. Zawarty w ciągu kod może definiować globalne zmienne i funkcje, przypisywać wartości globalnym zmiennym, ale nie może modyfikować lokalnych zmiennych funkcji wywołującej funkcję `eval()`. Zatem lokalna optymalizacja nie obejmuje takiego kodu.

Termin „bezpośrednia ewaluacja" oznacza wywołanie funkcji `eval()` w wyrażeniu wykorzystującym niekwalifikowaną nazwę „eval" (która na pierwszy rzut oka wygląda jak zarezerwowane słowo). Bezpośrednio wywołana funkcja `eval()` wykorzystuje zmienne stosowane w wywołującym ją

Kup ksi k Pole ksi k

kodzie. We wszystkich innych wywołaniach, na przykład pośrednich, wykorzystywane są globalne zmienne i nie można odczytywać, zapisywać ani definiować lokalnych zmiennych funkcji. W wywołaniach bezpośrednich i pośrednich nowe zmienne można definiować wyłącznie za pomocą instrukcji `var`. Instrukcje `let` i `const` użyte w ciągu tworzą lokalne zmienne i stałe i nie wpływają na środowisko nadrzędnej funkcji ani środowisko globalne.

Ilustruje to poniższy kod:

```
const geval = eval;              // Inna nazwa, przeznaczona do globalnego wywołania funkcji eval().
let x = "global", y = "global";  // Dwie zmienne globalne.
function f() {                   // Ta funkcja wywołuje funkcję eval() lokalnie.
  let x = "local";               // Definicja lokalnej zmiennej.
  eval("x += 'changed';");       // Funkcja eval() przypisuje wartość zmiennej lokalnej.
  return x;                      // Zwrócenie zmodyfikowanej zmiennej lokalnej.
}
function g() {                   // Ta funkcja wywołuje funkcję eval() globalnie.
  let y = "local";               // Lokalna zmienna.
  geval("y += 'changed';");      // Pośrednie przypisanie wartości zmiennej globalnej.
  return y;                      // Zwrócenie niezmodyfikowanej zmiennej lokalnej.
}
console.log(f(), x); // Zmodyfikowana zmienna lokalna. Wyświetlany wynik: "localchanged global".
console.log(g(), y); // Zmodyfikowana zmienna globalna. Wyświetlany wynik "local globalchanged".
```

Zwróć uwagę, że możliwość globalnego wywoływania funkcji `eval()` nie oznacza jedynie przystosowania jej do wymagań optymalizatora. W rzeczywistości jest to niezwykle przydatna funkcjonalność umożliwiająca uruchamianie fragmentów kodu tak, jakby były niezależnymi skryptami operującymi na najwyższym poziomie kodu. Jak wspomniałem na początku podrozdziału, rzadko pojawia się potrzeba uruchamiania kodu w postaci ciągu znaków. Jeżeli jednak jest to konieczne, należy w tym celu wywoływać funkcję `eval()` globalnie, a nie lokalnie.

4.12.3. Ścisłe wywołanie funkcji eval()

Tryb ścisły (patrz punkt 5.6.3) nakłada kolejne ograniczenia na funkcję `eval()`, a nawet identyfikator „eval". Jeżeli funkcja zostanie wywołana w trybie ścisłym lub ciąg umieszczony w jej argumencie będzie rozpoczynał się od dyrektywy `"use strict"`, to zawarty w nim kod zostanie wykonany w lokalnym, prywatnym środowisku. Oznacza to, że będzie mógł odczytywać i przypisywać wartości lokalnym zmiennym, ale nie będzie mógł definiować lokalnych zmiennych ani funkcji.

Co więcej, tryb ścisły sprawia, że funkcja `eval()` jeszcze bardziej przypomina operator, a „eval" staje się niemal słowem zarezerwowanym. Nie można zdefiniować funkcji o nazwie `eval()` ani zmiennej, parametru funkcji i parametru instrukcji `catch` o nazwie „eval".

4.13. Inne operatory

W języku JavaScript dostępnych jest jeszcze kilka innych operatorów, opisanych w poniższych punktach.

Kup ksi k Pole ksi k

4.13.1. Operator warunkowy (?:)

Operator warunkowy jest jedynym **operatorem trójargumentowym**. Opisuje się go jako `?:`, ale nie używa w takiej postaci w kodzie. Ponieważ ma on trzy operandy, pierwszy umieszczany jest przed znakiem `?`, drugi pomiędzy znakami `?` a `:`, a trzeci po znaku `:`, jak niżej:

```
x > 0 ? x : -x        // Wartość bezwzględna zmiennej x.
```

Operandy tego operatora mogą być dowolnych typów. Pierwszy jest interpretowany jako wartość logiczna. Jeżeli jest prawdziwa, wyliczana i zwracana jest wartość drugiego operandu, a w przeciwnym razie trzeciego operandu. Zawsze wyliczana jest wartość tylko drugiego lub trzeciego operandu, nigdy obu.

Ten sam efekt można osiągnąć za pomocą instrukcji `if` (patrz punkt 5.3.1), jednak operator `?:` jest bardziej zwięzły. Poniżej przedstawiony jest typowy przypadek, w którym operator sprawdza, czy dana zmienna została zadeklarowana (tj. czy ma wartość prawdziwą). Jeżeli tak, zwraca jej wartość. W przeciwnym razie operator zwraca wartość domyślną:

```
greeting = "dzień " + (username ? username : "dobry");
```

Jest to bardziej zwięzły odpowiednik poniższej instrukcji `if`:

```
greeting = "dzień ";
if (username) {
  greeting += username;
} else {
  greeting += "dobry";
}
```

4.13.2. Pierwszy zdefiniowany (??)

Operator `??` zwraca wartość pierwszego zdefiniowanego operandu. Jeżeli lewy operand ma wartość inną niż `null` lub `undefined`, operator ją zwraca. W przeciwnym razie zwraca wartość prawego operandu. Operator ten, podobnie jak `&&` i `||`, działa na zasadzie krótkiego zwarcia, tj. wylicza wartość drugiego operandu tylko wtedy, gdy pierwszy ma wartość `null` lub `undefined`. Jeżeli wyrażenie `a ?? b` nie wywołuje efektów ubocznych, jest ono równoważne następującemu wyrażeniu:

```
(a !== null && a !== undefined) ? a : b
```

Operator `??` stanowi użyteczną alternatywę dla operatora `||` (patrz punkt 4.10.2), gdy trzeba wybrać pierwszy zdefiniowany, a nie pierwszy prawdziwy operand. Choć operator `||` jest z założenia operatorem LUB, można go w wyrażeniu takim jak poniższe używać do wybierania pierwszego niefałszywego operandu:

```
// Jeżeli zmienna maxWidth jest prawdziwa, użyj jej. W przeciwnym razie poszukaj jej
// w obiekcie preferences. Jeżeli nie jest prawdziwa, użyj wpisanej stałej.
let max = maxWidth || preferences.maxWidth || 500;
```

Problem z takim idiomatycznym użyciem operatora polega na tym, że liczba zero, pusty ciąg znaków i wartość logiczna `false` są traktowane jako wartości fałszywe, które w określonych warunkach mogą być całkowicie poprawne. W powyższym przykładzie zmienna `maxWidth` o wartości 0 zostanie pominięta. Jeżeli natomiast operator `||` zostanie zmieniony na `??`, powstanie wyrażenie, w którym liczba zero będzie poprawną wartością:

Kup książkę Poleć książkę

```
// Jeżeli zmienna maxWidth jest prawdziwa, użyj jej. W przeciwnym razie poszukaj jej
// w obiekcie preferences. Jeżeli nie jest zdefiniowana, użyj wpisanej stałej.
let max = maxWidth ?? preferences.maxWidth ?? 500;
```

Poniżej przedstawionych jest kilka dodatkowych przykładów pokazujących, jak działa operator `??`, gdy pierwszy operand ma wartość fałszywą. Jeżeli taki operand jest zdefiniowany, operator zwraca go. Drugi operand jest wyliczany i zwracany tylko wtedy, gdy pierwszy ma wartość `null` lub `undefined`:

```
let options = { timeout: 0, title: "", verbose: false, n: null };
options.timeout ?? 1000     // => 0: wartość zdefiniowana w obiekcie.
options.title ?? "Untitled" // => "": wartość zdefiniowana w obiekcie.
options.verbose ?? true     // => false: wartość zdefiniowana w obiekcie.
options.quiet ?? false      // => false: właściwość nie jest zdefiniowana.
options.n ?? 10             // => 10: właściwość ma wartość null.
```

Zwróć uwagę, że gdyby zamiast operatora `??` był użyty operator `||`, wyniki powyższych wyrażeń wykorzystujących właściwości `timeout`, `title` i `verbose` byłyby inne.

Operator `??` jest podobny do operatorów `&&` i `||` i jego priorytet nie jest ani wyższy, ani niższy. Dlatego w wyrażeniach wykorzystujących te operatory należy za pomocą nawiasów określać, które operacje mają być wykonane w pierwszej kolejności:

```
(a ?? b) || c    // Najpierw ??, potem ||.
a ?? (b || c)    // Najpierw ||, potem ??.
a ?? b || c      // SyntaxError: wymagane użycie nawiasów.
```

Operator `??` został wprowadzony w wersji języka ES2020 i od początku 2020 r. jest obsługiwany we wstępnych wersjach większości najpopularniejszych przeglądarek. Formalnie jest nazywany operatorem „zerowołączącym". Unikam jednak tego terminu, ponieważ operator wybiera jeden z operandów, ale nie łączy ich w żaden widoczny sposób.

4.13.3. Operator typeof

Operator `typeof` jest jednoargumentowy i umieszcza się go przed operandem, który może być dowolnego typu. Zwracanym wynikiem jest ciąg znaków opisujący typ operandu. Tabela 4.3 przedstawia wyniki operatora `typeof` użytego ze wszystkimi wartościami dostępnymi w języku JavaScript.

Operator `typeof` można stosować w wyrażeniach takich jak poniższe:

```
// Jeżeli zmienna value jest typu string, umieść ją wewnątrz apostrofów.
// W przeciwnym razie przekształć w ciąg znaków.
(typeof value === "string") ? "'" + value + "'" : value.toString()
```

Zwróć uwagę, że jeżeli operand ma wartość `null`, operator `typeof` zwraca wartość `"object"`. Jeżeli trzeba odróżnić wartość `null` od obiektu, należy ją sprawdzić jawnie.

Funkcje w języku JavaScript są specjalnego rodzaju obiektami, ale operator `typeof` odróżnia je od obiektów do tego stopnia, że rezerwuje dla nich osobną zwracaną wartość.

Kup ksi k Pole ksi k

Tabela 4.3. Wyniki zwracane przez operator typeof

x	typeof x
`undefined`	`"undefined"`
`null`	`"object"`
`true` lub `false`	`"boolean"`
Dowolna liczba lub `NaN`	`"number"`
Dowolna liczba `BigInt`	`"bigint"`
Dowolny ciąg znaków	`"string"`
Dowolny symbol	`"symbol"`
Dowolna funkcja	`"function"`
Dowolny obiekt inny niż funkcja	`"object"`

Ponieważ operator `typeof` użyty z obiektem lub tablicą (ale nie funkcją) zwraca wartość `"object"`, wystarczy w jakiś sposób odróżniać obiekt tylko od prymitywnych typów. Aby odróżnić jedną klasę od innej, trzeba zastosować inną technikę, na przykład użyć operatora `instanceof` (patrz punkt 4.9.4), atrybutu klasy (punkt 14.4.3) lub właściwości konstruktora (punkt 9.2.2 i podrozdział 14.3).

4.13.4. Operator delete

Operator `delete` jest jednoargumentowy i służy do usuwania właściwości obiektu lub elementu tablicy wskazanego za pomocą operandu. Podobnie jak operator przypisania, inkrementacji i dekrementacji, obszar `delete` jest zazwyczaj stosowany w celu wywołania efektu ubocznego, a nie uzyskania wartości. Poniżej jest przedstawionych kilka przykładów:

```
let o = { x: 1, y: 2}; // Początkowy obiekt.
delete o.x;            // Usunięcie jednej z właściwości obiektu.
"x" in o               // => false: dana właściwość już nie istnieje.
let a = [1,2,3];       // Początkowa tablica.
delete a[2];           // Usunięcie ostatniego elementu tablicy.
2 in a                 // => false: element o indeksie 2 już nie istnieje.
a.length               // => 3: zwróć uwagę, że długość tablicy nie zmieniła się.
```

Zwróć uwagę, że usuniętej w ten sposób właściwości lub elementowi tablicy nie jest jedynie przypisywana wartość `undefined`. Usunięta właściwość przestaje istnieć. Próba odwołania się do niej skutkuje uzyskaniem wartości `undefined`, ale za pomocą operatora `in` (patrz punkt 4.9.3) można sprawdzić, czy właściwość w ogóle istnieje. W miejscu usuniętego elementu tablicy powstaje luka, ale długość tablicy nie zmienia się. Wynikowa tablica jest **rozrzedzona** (patrz podrozdział 14.1).

Argumentem operatora `delete` powinna być *l*-wartość. Operator podejmuje próbę jej usunięcia i w przypadku powodzenia zwraca `true`. Jeżeli typ jest inny, operator nie wykonuje żadnej operacji i zwraca wartość `true`. Jednak niektórych właściwości, na przykład niekonfigurowalnych (patrz podrozdział 14.1) nie da się usunąć.

Kup ksi k
Pole ksi k

Jeżeli w trybie ścisłym operand jest niekwalifikowanym identyfikatorem, na przykład zmienną, funkcją lub parametrem funkcji, operator `delete` zgłasza wyjątek SyntaxError. Kod jest poprawny tylko wtedy, gdy operand jest wyrażeniem dostępu do właściwości (patrz podrozdział 4.4). Ponadto w trybie ścisłym operator zgłasza wyjątek TypeError przy próbie usunięcia niekonfigurowalnej (czyli nieusuwalnej) właściwości. Jeżeli tryb ścisły nie jest stosowany, wyjątki nie są zgłaszane, a operator `delete` po prostu zwraca wartość `false` oznaczającą, że operandu nie można usunąć.

Poniżej przedstawionych jest kilka przykładów użycia operatora `delete`:

```
let o = {x: 1, y: 2};
delete o.x;   // Usunięcie jednej z właściwości obiektu. Wynik: true.
typeof o.x;   // Właściwość nie istnieje. Wynik: true.
delete o.x;   // Próba usunięcia nieistniejącej właściwości. Wynik: true.
delete 1;     // To wyrażenie nie ma sensu, ale jego wynikiem jest wartość true.
// Zmiennej nie można usunąć. Wynik: true lub wyjątek SyntaxError w trybie ścisłym.
delete o;
// Nieusuwana właściwość. Wynik: false lub wyjątek TypeError w trybie ścisłym.
delete Object.prototype;
```

Z operatorem `delete` spotkasz się ponownie w podrozdziale 6.4.

4.13.5. Operator await

Operator `await` został wprowadzony w wersji języka ES2017, aby programowanie asynchroniczne w JavaScripcie było bardziej naturalne. Jego działanie poznasz w rozdziale 13. W uproszczeniu jedynym operandem operatora `await` jest promesa, czyli obiekt reprezentujący asynchroniczną operację. Operator sprawia, że kod czeka na zakończenie wykonania promesy, jednak nie blokuje swojego działania ani nie wstrzymuje innych asynchronicznych operacji. Wynikiem operatora jest wartość opisująca spełnienie promesy. Co istotne, można go stosować tylko z funkcjami zadeklarowanymi jako asynchroniczne za pomocą słowa kluczowego `async`. Szczegółowe informacje znajdziesz we wspomnianym wyżej rozdziale 13.

4.13.6. Operator void

Operator `void` jest jednoargumentowy, a jego operand może być dowolnego typu. Jest to nietypowy i rzadko stosowany operator, który wylicza wartość operandu, po czym ją odrzuca i zwraca wartość `undefined`. Z tego względu stosowanie go jest uzasadnione tylko wtedy, gdy operand wywołuje efekty uboczne.

Operator `void` jest tak niezwykły, że trudno jest podać przykład jego praktycznego użycia. Jednym z nich może być funkcja, która nie zwraca wyniku, zdefiniowana przy użyciu składni ze strzałką (patrz punkt 8.1.3) i jednym wyrażeniem. Jeżeli takie wyrażenie ma być wyliczone wyłącznie w celu wywołania efektu ubocznego, najprościej jest umieścić ciało funkcji wewnątrz nawiasów klamrowych. Ewentualnie można użyć operatora `void`:

```
let counter = 0;
const increment = () => void counter++;
increment()   // => undefined
counter       // => 1
```


Kup ksi k Pole ksi k

4.13.7. Operator przecinek (,)

Przecinek jest operatorem dwuargumentowym, a jego operandy mogą być dowolnych typów. Operator wylicza wartości obu operandów i zwraca wartość prawego. Zatem wynikiem poniższego kodu:

```
i=0, j=1, k=2;
```

jest liczba 2, a kod jest równoważny następującemu:

```
i = 0; j = 1; k = 2;
```

Wyrażenie pierwsze z lewej jest wyliczane zawsze, ale jego wynik jest odrzucany. Oznacza to, że stosowanie przecinka jest uzasadnione jedynie wtedy, gdy wyrażenie to wywołuje efekty uboczne. Jedyną konstrukcją, w której przecinek jest powszechnie stosowany, jest pętla `for` (patrz punkt 5.4.3) wykorzystująca kilka zmiennych:

```
// Pierwszy przecinek jest częścią składni instrukcji let. Drugi przecinek jest operatorem.
// Dzięki niemu można scalić dwa wyrażenia (i++ oraz j--) i umieścić w instrukcji
// operującej na jednym wyrażeniu (pętli for).
for(let i=0,j=10; i < j; i++,j--) {
  console.log(i+j);
}
```

4.14. Podsumowanie

W tym rozdziale zostało poruszonych wiele różnych tematów z odniesieniami do mnóstwa fragmentów, które zapewne przeczytasz w przyszłości ponownie, kontynuując naukę programowania w języku JavaScript. Poniżej wymienione są najważniejsze zagadnienia, o których warto pamiętać:

- Wyrażenia to frazy programu.
- Każde wyrażenie ma wartość.
- Wyrażenia mogą oprócz zwracania wartości wywoływać efekty uboczne (na przykład przypisywać zmiennej nową wartość).
- Proste wyrażenia, takie jak literały czy odwołania do zmiennych i właściwości, można łączyć ze sobą za pomocą operatorów i tworzyć w ten sposób bardziej złożone wyrażenia.
- W języku JavaScript są dostępne m.in. operatory arytmetyczne, relacyjne, logiczne, przypisujące i bitowe. Jest też trójargumentowy operator warunkowy.
- Operator + wykorzystuje się do sumowania wartości, jak również do łączenia ze sobą ciągów znaków.
- Operatory logiczne `&&` i `||` działają na zasadzie „krótkiego zwarcia", tzn. mogą wyliczać wartość tylko jednego z dwóch operandów. Do stosowania popularnych konstrukcji składniowych języka JavaScript wymagana jest znajomość działania tych operatorów.

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Dowiedz się wszystkiego, co musisz wiedzieć o JavaScripcie!

JavaScript jest wykorzystywanym w wielu celach uniwersalnym językiem ogólnego przeznaczenia — wysokopoziomowym, dynamicznym, przygotowanym do kodowania obiektowego i funkcjonalnego. Zasadniczo służy do programowania sieci WWW: kod JavaScript występuje w zdecydowanej większości stron internetowych. To jednak tylko jedno z jego możliwych zastosowań. Wszystko to sprawia, że JavaScript jest najbardziej rozpowszechnionym językiem w historii programowania. Pojawienie się platformy Node.js spowodowało, że stał się wszechstronniejszy, a jego popularność wśród programistów jeszcze wzrosła.

To kolejne wydanie wyczerpującego, kompleksowego przewodnika po języku JavaScript oraz jego najważniejszych klienckich i serwerowych interfejsach API. Książka jest przeznaczona dla programistów, którzy chcą nauczyć się JavaScriptu lub udoskonalić swoje umiejętności tworzenia kodu. Ujęto w niej wersję ES2020 tego języka. Poza praktycznymi wyjaśnieniami dotyczącymi jego struktur i ich stosowania znalazły się tu liczne przykłady, wskazówki i porady. To wydanie zawiera nowe rozdziały poświęcone klasom, modułom, iteratorom, generatorom, promesom i instrukcjom async/await. W efekcie powstał całościowy obraz ekosystemu JavaScript wraz z opisem jego zawiłości, potencjalnych problemów i najlepszych metod radzenia sobie z nimi.

W książce między innymi:

- podstawy języka i jego podstawowe elementy
- struktury danych, wyrażenia regularne, format JSON
- standardowa biblioteka JavaScript
- przetwarzanie dokumentów i grafiki
- obsługa sieci, pamięci i wątków
- środowisko Node oraz profesjonalne narzędzia i rozszerzenia języka

David Flanagan od ćwierćwiecza programuje w JavaScripcie i pisze o tym języku. Jest doświadczonym inżynierem oprogramowania, pracuje w VMware. Mieszka z rodziną na północno-zachodnim wybrzeżu Stanów Zjednoczonych, gdzieś pomiędzy Seattle a kanadyjskim Vancouver.