



**REGEX**

Tomasz Sochacki

# JavaScript

WYRAŻENIA **REGULARNE** DLA PROGRAMISTÓW

**Helion** 

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

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

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

Opieka redakcyjna: Ewelina Burska

Projekt okładki: Studio Gravite/Olsztyn

Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/wyrzsp>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-3892-0

Copyright © Helion 2018

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

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

# Spis treści

<b>Rozdział 1. Na początek nieco teorii .....</b>	<b>7</b>
Czym są wyrażenia regularne? .....	7
Dla kogo przeznaczona jest książka .....	8
Nie ucz się regexp na pamięć! .....	9
Kilka słów o Unicode .....	10
Unicode to nie UTF! .....	11
Kodowanie znaków UTF-8 .....	12
Kodowanie znaków UTF-16 .....	13
Kodowanie znaków i BOM .....	15
<b>Rozdział 2. Składnia wyrażeń regularnych w JavaScriptcie .....</b>	<b>17</b>
Dopasowanie dosłowne .....	18
Znaki specjalne i sekwencja ucieczki .....	20
Dopasowanie znaków niedrukowanych i białych .....	23
Klasy znaków .....	25
Zakresy znaków w klasach .....	28
Predefiniowane znaki i zakresy znaków .....	31
Dopasowanie cyfry .....	31
Dopasowanie „wyrazu” .....	32
Problem sekwencji negujących wewnątrz klas znaków .....	34
Białe znaki .....	36
Dopasowanie znaku NULL .....	37
Analiza granicy wyrazu .....	38
Zestawienie predefiniowanych sekwencji znakowych .....	40
Grupy i odwołania wsteczne .....	41
Grupowanie z wykorzystaniem alternatywy .....	43
Grupy nieprzechwytyjące .....	46
Kwantyfikatory — podstawy .....	48
Kwantyfikatory skrótowe .....	51
Kwantyfikatory zachłanne i niezachłanne .....	53

Wyprzedzenia pozytywne i negatywne .....	56
Granice dopasowania początku i końca .....	58
Flagi we wzorcach wyrażeń regularnych .....	60
Flaga case insensitive (i) .....	61
Flaga dopasowania globalnego (g) .....	64
Flaga multiline (m) .....	67
Flaga unicode (u) .....	68
Flaga unicode (u) — poprawne rozpoznawanie wszystkich znaków Unicode .....	69
Flaga sticky (y) .....	73
Bezpieczne stosowanie sekwencji ucieczki .....	76
Granice dopasowania .....	77
Kwantyfikatory .....	78
Symbole specjalne wewnątrz klasy znaków .....	79
Inne symbole specjalne .....	81
<b>Rozdział 3. Praca z wyrażeniami regularnymi w JavaScriptcie .....</b>	<b>83</b>
Tworzenie wyrażeń regularnych .....	83
Sekwencje ucieczki w konstruktorze RegExp .....	84
Definiowanie flag podczas tworzenia wzorców .....	85
Tworzenie wzorców z innych wzorców .....	86
Komentowanie wzorców regexp .....	88
Omówienie właściwości obiektu RegExp .....	90
Obsługa flag .....	90
Znakowa reprezentacja wzorca regexp .....	91
Właściwości ostatnio użytego obiektu RegExp .....	91
Metoda RegExp.prototype.test .....	94
Obsługa flag case insensitive i unicode .....	94
Problematyczne włączanie flag global i sticky .....	95
Metoda RegExp.prototype.exec .....	97
Aktywna flaga global lub sticky .....	98
Metoda exec() i właściwość lastIndex .....	99
Różnica między index i lastIndex .....	101
Wzorzec zawierający grupy przechwytyjące i warunkowe .....	102
Użycie metody exec() w pętli .....	104
Metoda String.prototype.search .....	106
Metoda search() w instrukcjach warunkowych .....	107
Kwantyfikator dowolnej liczby powtórzeń .....	108
Flagi global i sticky w metodzie search() .....	109

---

Metoda <code>String.prototype.match</code> .....	110
Współpraca z flagą <code>global</code> .....	110
Aktywacja flagi <code>sticky</code> .....	111
Analiza dopasowanych grup przechwytyjących .....	113
Metoda <code>String.prototype.replace</code> .....	114
Dostęp do właściwości ostatniego obiektu <code>RegExp</code> .....	114
Nowy ciąg jako funkcja zwrotna .....	116
Argumenty funkcji zwrotnej i ich wykorzystanie .....	117
Grupy przechwytyjące jako dopasowania alternatywne .....	119
Wyrażenie regularne z flagą <code>sticky</code> .....	120
Metoda <code>String.prototype.split</code> .....	121
Wyrażenie regularne jako separator .....	121
Przechwycenie separatora dzielącego ciąg znakowy .....	122
Analiza ciągów znakowych bez użycia wyrażeń regularnych .....	124
Metoda <code>startsWith</code> .....	125
Metoda <code>endsWith</code> .....	126
Metoda <code>includes</code> .....	126
Metoda <code>indexOf()</code> .....	127
Metoda <code>lastIndexOf</code> .....	128
Praca ze znakami Unicode .....	129
Tworzenie ciągów znakowych Unicode .....	130
Metoda <code>charCodeAt</code> .....	131
Metoda <code>codePointAt</code> .....	132
Metody <code>fromCharCode</code> i <code>fromCodePoint</code> .....	133
<b>Rozdział 4. Przykładowe wyrażenia regularne z omówieniem .....</b>	<b>135</b>
Usuwanie białych znaków z początku i końca ciągu .....	135
Sprawdzenie, czy ciąg zawiera wyłącznie znaki alfanumeryczne .....	136
Walidacja liczby całkowitej w systemie dziesiętnym .....	138
Walidacja liczby z częścią ułamkową w systemie dziesiętnym .....	141
Walidacja liczby w systemie wykładniczym .....	142
Liczby w systemie dwójkowym i szesnastkowym .....	145
Weryfikacja kolorów w zapisie HEX, RGB i HSL .....	148
Sprawdzenie, czy liczba jest podzielna przez 2, 4 lub 5 .....	151
Czy liczba zawiera się we wskazanym zakresie .....	153
Analiza kodu pocztowego .....	155
Analiza numeru telefonu .....	158
Usuwanie przedrostków z ciągu zawierającego imię i nazwisko .....	160
Ciąg złożony z unikalnych znaków .....	161

Zamiana polskich znaków diakrytycznych na litery łacińskie .....	163
Usuwanie znaczników HTML .....	164
Zastępowanie znaczników HTML .....	165
Cenzura, czyli modyfikacja wskazanych wyrazów zabronionych .....	167
Wykrywanie powtórzeń zlokalizowanych w określonej odległości znakowej .....	168
Usuwanie komentarzy w kodzie aplikacji .....	171
Analiza plików INI według określonych reguł .....	173
Analiza poprawnego zapisu daty .....	176
Analiza poprawności czasu w różnych formatach .....	179
Weryfikacja poprawności hasła tworzonego podczas rejestracji .....	180
Analiza adresu URL i podział na elementy składowe .....	182
Walidacja numeru karty kredytowej z algorytmem Luhna .....	185
Weryfikacja poprawności zapisu liczb rzymskich .....	188
Analiza nazw i rozszerzeń plików .....	193
Walidacja adresu e-mail .....	195
Walidacja poprawności adresu IP (ipv4) .....	199
Walidacja poprawności MAC Address .....	201
Weryfikacja numeru PESEL .....	202
Weryfikacja numeru dowodu osobistego .....	204
Weryfikacja numeru NIP .....	206
Weryfikacja numeru REGON .....	208
Weryfikacja numeru ISBN .....	210
Weryfikacja europejskiego numeru płatnika VAT .....	214
Walidacja imienia i nazwiska, czyli kiedy lepiej nie stosować regexp .....	216
<b>Rozdział 5. Przyszłość wyrażen regularnych w JavaScriptcie .....</b>	<b>219</b>
Jak wygląda proces tworzenia standardów ECMAScript .....	219
Nowa flaga dotAll .....	220
Nazwane grupy przechwytyjące .....	222
Metoda match i exec .....	222
Zmodyfikowana metoda String.prototype.replace .....	224
Odwołania wsteczne wewnątrz wyrażenia regularnego .....	224
Cofnięcia pozytywne i negatywne .....	225
Grupy znakowe Unicode .....	227
Słowo końcowe .....	229
<b>Skorowidz .....</b>	<b>231</b>

## Rozdział 5.

# Przyszłość wyrażeń regularnych w JavaScriptcie

Wyrażenia regularne w JavaScriptcie są obsługiwane w znacznie mniejszym zakresie niż w innych językach, np. w Javie. W ostatnim czasie komitet normalizacyjny TC39 zaczyna jednak coraz bardziej rozszerzać możliwości regexp, a w niedługiej przyszłości planowane są kolejne zmiany.

Ten rozdział został poświęcony omówieniu planowanych nowych funkcjonalności wyrażeń regularnych na podstawie materiałów publikowanych przez komitet normalizacyjny. Można z nich korzystać po włączeniu w przeglądarce Chrome flagi *javascript-harmony* odpowiedzialnej za aktywowanie eksperymentalnych elementów silnika JavaScript:

```
chrome://flags/#enable-javascript-harmony
```

Niniejszy rozdział opisuje planowane zmiany w odniesieniu do ich eksperymentalnej implementacji w przeglądarce Chrome. Omawiane w tym rozdziale elementy są obecnie (stan na listopad 2017 r.) w stage 3, co oznacza, że z bardzo dużym prawdopodobieństwem znajdują się w kolejnym wydaniu standardu ECMAScript 2018.

## Jak wygląda proces tworzenia standardów ECMAScript

Na początku należy zaznaczyć, że JavaScript nie jest określeniem tożsamym z ECMAScript. Standard ES jest punktem wyjścia do tworzenia nowych wersji silników JavaScript. Przykładem może być chociażby kwestia obiektów globalnych w JavaScriptcie. W silnikach obsługiwanych przez przeglądarki internetowe dostępny jest np. obiekt `window`, którego nie ma jednak w `node.js`, działającym po stronie serwerowej.

Rozwojem standardu ECMAScript zajmuje się komitet normalizacyjny TC39. W jego skład wchodzi głównie przedstawiciele firm tworzących przeglądarki internetowe. W regularnych odstępach czasu spotykają się i dyskutują o nowych propozycjach zmian w standardzie, które później będą implementować w poszczególnych przeglądarkach.

Cały proces od pomysłu do wdrożenia rozwiązania w standardzie ECMAScript obejmuje pięć etapów.

- **Stage 0** — w tej fazie dyskutuje się o pomysłe bez jakichkolwiek analiz jego implementacji. Najczęściej jeden z członków komitetu zgłasza jakąś propozycję i jeśli zostanie ona przyjęta, to wchodzi do kolejnego etapu.
- **Stage 1** — po wstępnej akceptacji pomysłu w etapie 1. jeden z członków komitetu przygotowuje dokładny opis proponowanej funkcjonalności wraz z przykładowymi algorytmami. Jeśli w tym momencie rozwiązanie zostanie przyjęte, to przejście do etapu 2. jest równoważne ze wstępną zgodą na włączenie danego pomysłu do docelowego standardu ECMAScript. Opisywane rozwiązania mogą się jednak jeszcze zmieniać na kolejnych etapach.
- **Stage 2** — na tym etapie proponowane rozwiązanie musi być już kompletne, to znaczy opisane językiem zgodnym ze stosowanym w standardzie ECMAScript, a wszelkie dalsze zmiany mogą być jedynie zmianami przyrostowymi lub poprawką błędów zgłoszonych na etapie testowania. Aby przejść do etapu 3., muszą zostać opracowane co najmniej dwie eksperymentalne implementacje.
- **Stage 3** — w tym momencie cała proponowana funkcjonalność jest już kompletna i w pełni opisana w standardzie. Ponadto są już gotowe eksperymentalne implementacje. Jest to etap, w którym ponownie dyskutuje się nad daną funkcjonalnością i podejmuje się ostateczną decyzję co do wpisania jej do standardu ECMAScript.
- **Stage 4** — ostatni etap, w którym gotowe już eksperymentalne implementacje poddaje się różnym testom. Jeśli wszystkie testy przechodzą prawidłowo i pomysł ma już pełną dokumentację, to znajdzie się on w najbliższym wydaniu nowego standardu ECMAScript. Rzeczywista implementacja w przeglądarkach i innych środowiskach może się jednak istotnie opóźnić w zależności od stopnia skomplikowania danego rozwiązania. Warto jednak śledzić rozwój transpilatora Babel, który często pozwala korzystać z nowych funkcjonalności, gdy są we wczesnych fazach, długo przed ich wdrożeniem do standardu ES.

## Nowa flaga dotAll

Znak kropki często jest definiowany jako dopasowanie dowolnego znaku. Jak może pa-miętasz z rozdziału 2., definicja ta nie jest do końca poprawna. Po pierwsze nie dopasowuje ona sekwencji oznaczającej nowy wiersz, a po drugie działa tylko w podstawowym zakresie znaków Unicode, czyli do punktu kodowego U+FFFF.

```
/^.$/.test( 'a' );           // true
/^.$/.test( '\n' );        // false
/^.$/.test( '\u{1F31B}' ); // false
```



Zwróć uwagę, że w powyższych przykładach zastosowaliśmy granice początku i końca ciągu, co skutkuje próbą dopasowania dokładnie jednego dowolnego znaku. Bez tych granic ostatecznie wywołanie metody `test` zakończyłoby się powodzeniem, gdyż jako dowolny znak zostałby dopasowany pierwszy surogat znaku (jak zwykle użyliśmy naszego symbolu półksiężyca).

Pytanie jednak, jak stworzyć wzorzec, który faktycznie dopasuje dowolny znak z pełnego zakresu Unicode. Po części problem ten udało nam się już rozwiązać, stosując sztuczkę z odpowiednim zakresem znaków:

```
 /^[s\S]$/.test( '\n' );           // true
 /^[s\S]$/.test( '\u{1F31B}' );   // false
 /^[s\S]$/.test( '\u{1F31B}' );   // true po włączeniu flagi unicode
```

Tworząc zakres znaków `[s\S]`, w rzeczywistości próbujemy dopasować dowolny znak, ale żeby uwzględnić również znaki spoza podstawowej platformy wielojęzycznej BMP, konieczne jest aktywowanie flagi `unicode`.

Metoda ta jest jednak mało czytelna, szczególnie dla osób nieznających dokładnie zasad tworzenia klas znaków. Ponadto, jeśli we wzorcu będziemy potrzebowali kilka razy dopasować dowolny znak, to powyższy zapis istotnie skomplikuje całe wyrażenie regularne.

Z pomocą przychodzi nowa flaga `dotAll` (`s`). Po jej włączeniu znak kropki dopasowuje wszystkie znaki Unicode do `U+FFFF`, łącznie ze znakiem nowego wiersza. Jeśli chcemy, aby kropka dopasowała znaki spoza zakresu BMP, to musimy dodatkowo włączyć również flagę `unicode`.

```
 /^.$/s.test( '\n' );           // true
 /^.$/s.test( '\u{1F31B}' );   // false
 /^.$/su.test( '\u{1F31B}' );  // true
```

Należy jednak pamiętać, że omawiane powyżej specjalne znaczenie kropki jest ważne tylko wtedy, gdy znak ten znajduje się poza nawiasami kwadratowymi, określającymi zakres znaków. Wewnątrz zakresu kropka jest dopasowywana w sposób dosłowny.

Po wprowadzeniu flagi `dotAll` zmienia się również zalecana kolejność definiowania flag w wyrażeniach regularnych, która jest następnie stosowana przy odczycie właściwości `flags` dla obiektu `RegExp`:

```
"gimsuy"
```

## Nazwane grupy przechwytyjące

Dotychczas, aby odwołać się do przechwyconych grup, musieliśmy znać ich dokładne numery, co przy bardziej rozbudowanych wzorcach wymagało dłuższej analizy. Szczególnie ostrożnie należy tworzyć zagnieżdżone grupy przechwytyjące, co omawialiśmy dokładnie w rozdziale 2.

Przeanalizujmy prosty przykład dopasowania daty zgodnej ze standardem najczęściej używanym w Polsce, czyli dzień-miesiąc-rok:

```
/(\d{2})-(\d{2})-(\d{4})/
```

Jeśli teraz chcielibyśmy odnieść się do poszczególnych grup, to musieliśmybyśmy dokładnie pamiętać kolejność, w jakiej zostały przechwycone. Problem ten rozwiązują nazwane grupy przechwytyjące, które pozwalają na inny zapis powyższego wyrażenia regularnego:

```
/(?<day>\d{2})-(?<month>\d{2})-(?<year>\d{4})/
```

Zapis ten na pewno jest dłuższy, ale jednocześnie zwiększa czytelność wzorca i ułatwia jego późniejszą analizę. Nie musimy w tym momencie tworzyć odpowiednich komentarzy informujących, czym są poszczególne grupy, wystarczy nadać im odpowiednio precyzyjne nazwy. Zauważ, że w tym wypadku grupę definiuje się w nieco odmienny sposób:

```
(?<nazwa>)
```

Pomiędzy znacznikami < i > należy określić nazwę grupy, stosując w tym celu znaki, które są dopuszczone podczas definiowania właściwości obiektów w JavaScriptcie. Nie należy tutaj stosować m.in. znaków spacji.

## Metoda match i exec

Przeanalizujmy stworzone przed chwilą wyrażenie regularne użyte wraz z metodami `String.prototype.match` oraz `RegExp.prototype.exec`:

```
const reg = /( ?<day>\d{2})-( ?<month>\d{2})-( ?<year>\d{4})/

const match = '12-05-2017'.match( reg );
match;
// ["12-05-2017", "12", "05", "2017", index: 0, input: "12-05-2017", groups: {...}]

const exec = reg.exec( '12-05-2017' );
exec;
// ["12-05-2017", "12", "05", "2017", index: 0, input: "12-05-2017", groups: {...}]
```

Zwróć uwagę, że w obu przypadkach korzystamy z wyrażenia regularnego, w którym nie ma aktywnej flagi *global*. Jeśli zostałaby ona włączona, to metoda `match` zwróciłaby tablicę z jednym elementem, którym byłby ciąg znakowy reprezentujący znalezione dopasowanie (w tym wypadku byłby on identyczny z ciągiem wejściowym).

Tablice zwrotne mają elementy identyczne jak omawiane w rozdziale 3. Zauważ jednak, że tablica jako obiekt posiada dodatkową właściwość `groups`, w której zapisane są wszystkie dopasowane grupy nazwane. Do grup tych możemy więc odnieść się na dwa sposoby:

```
match[1];           // "12"
match.groups.day;  // "12"

exec[1];            // "12"
exec.groups.day;   // "12"
```

Nie należy więc obawiać się, że wprowadzenie grup nazwanych może zaburzyć działanie istniejących aplikacji. W języku JavaScript jednym z ważniejszych problemów jest konieczność zachowania zgodności wstecz, w związku z czym zdecydowano się na dodanie kolejnego elementu, bez usuwania dotychczasowych.

Tworząc wyrażenia regularne z grupami przechwytyjącymi, należy zdecydować się, czy w danym przypadku stosujemy grupy nazwane, a jeśli tak, to używać ich dla wszystkich grup. Zwrotny obiekt `groups` zawiera bowiem tylko grupy, którym nadano nazwę:

```
const reg = /(<digit>\d)([A-Z])/;
const match = '5A'.match( reg );

match.groups; // {digit: "5"}
// ale:
match[1]; // "5"
match[2]; // "A"
```

Stosując grupę nazwaną, możemy mieć jednak pewność, że obiekt `groups` zawsze będzie zawierał taką właściwość, nawet jeśli grupie zostanie przypisany kwantyfikator wystąpienia warunkowego (przyjrzyj się dokładnie położeniu kwantyfikatora ?):

```
const reg = /(<digit>\d?)([A-Z])/;
const match = 'A'.match( reg );
match.groups; // {digit: ""}

const reg1 = /(<digit>\d)?([A-Z])/;
const match1 = 'A'.match( reg1 );
match1.groups; // {digit: undefined}
```

Bardzo podobnie zachowuje się metoda `RegExp.prototype.exec`:

```
const reg = /(<digit>\d?)([A-Z])/;
const exec = reg.exec( 'A' );
exec.groups; // {digit: ""}

const reg1 = /(<digit>\d)?([A-Z])/;
const exec1 = reg1.exec( 'A' );
exec1.groups; // {digit: undefined}
```

## Zmodyfikowana metoda `String.prototype.replace`

Podobnie jak metody `match` i `exec`, również i metoda `String.prototype.replace` może dysponować dodatkowym obiektem `groups`, jeśli w wyrażeniu regularnym znajdują się grupy nazwane. Obiekt ten jest dostępny w funkcji zwrotnej (użytej jako drugi argument metody `replace`) jako ostatni jej parametr:

```
const reg = /(?!<month>\d{2})-(?!<year>\d{4})/

'12-2017'.replace( reg, ( match, p1, p2, offset, string, groups ) => {
  // Odwracamy kolejność:
  return `${groups.year}-${groups.month}`;
} ); // "2017-12"
```

Można oczywiście zastosować również zapis wykorzystujący destrukuryzację obiektu `groups`:

```
'12-2017'.replace( reg, ( match, p1, p2, offset, string, { month, year } ) => {
  // Odwracamy kolejność:
  return `${year}-${month}`;
} ); // "2017-12"
```

Zauważ, że w obu przypadkach dostępne są również dopasowania do grup przechwytyjących zapisane w zmiennych `p1` i `p2`. Jest to konieczne dla zachowania zgodności wstecz. Wewnątrz funkcji zwrotnej warto jednak trzymać się jednej konwencji, co zapewni czytelność kodu, łatwość jego utrzymania i wprowadzania zmian. Należy również pamiętać, że obiekt z grupami nazwanymi dostępny jest za zmiennymi `offset` oraz `string`.

## Odwołania wsteczne wewnątrz wyrażenia regularnego

Tworząc nazwane grupy przechwytyjące, można się do nich odwoływać również w samych wyrażeniach regularnych. W tym celu należy zastosować składnię z sekwencją `\k`, a po niej należy wskazać grupę, do której chcemy się odnieść:

```
(?!<nazwa>) ... \k<nazwa>
```

Przeanalizujmy poniższy przykład, w którym chcemy sprawdzić, czy ciąg znakowy zawiera dowolną cyfrę, następnie co najmniej jedną małą literę i ponownie tę samą cyfrę. Co prawda w tak prostym przykładzie w praktyce nie ma sensu stosowanie grup nazwanych, ale w bardziej złożonych wzorcach jest to dobra praktyka, ułatwiająca analizę wyrażenia regularnego.

```
const reg = /(?!<digit>\d)[a-z]+\k<digit>/

reg.test( '5abc5' ); // true
reg.test( '5abc3' ); // false
```

Wszystkie dopasowane grupy nadal są dostępne również poprzez odwołania numeryczne, czyli sekwencję `\N`. Grupom nadawane są kolejne numery zgodnie z występowaniem kolejnych nawiasów grupujących od lewej do prawej. Oczywiście pomijane są przy tym grupy nieprzechwytyjące, ale każda grupa nazwana zawsze stanowi grupę przechwytyjącą, zapamiętaną pod wskazaną nazwą oraz kolejnym numerem.

```
const reg = /(?!<digit>\d)\1\k<digit>/
reg.test( '555' ); // true
reg.test( '543' ); // false
```

Stosując nazwane grupy przechwytyjące, trzeba pamiętać, że istotna jest wielkość znaków użytych do opisanía grupy:

```
const reg = /(?!<digit>\d)\1\k<DIGIT>/
// SyntaxError: Invalid named capture referenced
```

## Cofnięcia pozytywne i negatywne

Do tej pory mogliśmy w JavaScriptcie korzystać z wyprzedzeń zarówno pozytywnych, jak i negatywnych, ale nie można było w sposób bezpośredni używać cofnięć, które od dawna dostępne są w wielu innych językach programowania.

Teraz ma się to zmienić i mamy otrzymać nowy element składniowy, tzw. *lookbehind assertions*. Definiowanie cofnięć wygląda następująco:

```
/(?<=...)/ <- dla cofnięć pozytywnych
/(?!...)/ <- dla cofnięć negatywnych
```

Co istotne, podobnie jak jest w przypadku wyprzedzeń, cofnięcia także nie są zwracane w dopasowanym ciągu znakowym. Daje nam to duże możliwości precyzyjnej analizy ciągów i zwracania od razu tylko dokładnie wyselekcjonowanych fragmentów, ograniczając tym samym konieczność ich dalszego przetwarzania (np. metodą `String.prototype.replace`).

Przeanalizujmy prosty przykład cofnięcia pozytywnego i negatywnego:

```
// $ === \u0024
// € === \u20AC

const price = '$100 €200';

price.match( /(?!<=\\u0024)\\d{3}/g ); // ["100"]
price.match( /(?!<=\\u20AC)\\d{3}/g ); // ["200"]

price.match( /(?!\\u0024)\\d{3}/g ); // ["200"]
price.match( /(?!\\u20AC)\\d{3}/g ); // ["100"]
```

W powyższych przykładach próbujemy dopasować liczbę trzycyfrową, przed którą występuje lub nie występuje odpowiednio symbol dolara lub euro.

Jako cofnięcie, podobnie jak przy wyprzedzeniach, można wskazać dowolne wyrażenie regularne, a więc również wyrażenie zawierające grupy przechwytyjące:

```
const price = '$100 €200';

// Grupa wewnątrz cofnięcia:
/(?!<=(\\u20AC))\\d{3}/.exec( price );
```

```
// ["200", "€", index: 6, input: "$100 €200"]
// Dodatkowa grupa poza cofnięciem:
/(?<=<(\u20AC))(\d{3})/.exec( price );
// ["200", "€", "200", index: 6, input: "$100 €200"]
```

Oczywiście również w tym przypadku można stosować grupy nazwane:

```
const p = /(?<=<(currency>\u20AC))(price>\d{3})/.exec( price );
p.groups.currency; // "€"
p.groups.price; // "200"
```

Grupy przechwytyjące stworzone wewnątrz cofnięcia są również dostępne poza nim w pozostałej części wyrażenia regularnego. Pozwala to na przykład stworzyć poniższy wzorzec, który z wejściowego ciągu zwraca liczby, które na początku i na końcu ograniczone są taką samą wielką literą alfabetu łacińskiego, lecz bez zwracania tych granicznych znaków:

```
'X123X A456A B234C'.match( /(?<=<([A-Z]))\d+(?=\1)/g ); // ["123", "456"]
```

Pewien problem pojawia się jednak, jeśli wewnątrz cofnięcia chcemy stosować odwołania do przechwyconych grup. Przeanalizujemy dwa poniższe przykłady:

```
'X_Xaaa Y_Ybbb X_Zccc'.match( /(?<=<([A-Z])_)\1[a-z]+/g );
// null
'X_Xaaa Y_Ybbb X_Zccc'.match( /(?<=\1_([A-Z]))\1[a-z]+/g );
// ["aaa", "bbb"]
```

Naszym celem było stworzenia wzorca zwracającego ciąg małych liter (aaa, bbb), przed którymi znajduje się ciąg składający się z wielkiej litery, znaku podkreślenia i ponownie tej samej wielkiej litery (np. X\_X, ale nie X\_Z).

Prawdopodobnie w pierwszym przypadku oczekiwaliśmy, że nasz wzorzec zadziała prawidłowo, gdyż wewnątrz cofnięcia stworzyliśmy podwzorzec przechwytyjący wielką literę, a następnie sprawdzający, czy występuje ona ponownie. Otrzymaliśmy jednak zwrótnie wartość null wskazującą na brak możliwego do znalezienia dopasowania.

Otóż błąd tkwi w tym, że wewnątrz cofnięć do przechwytywanych grup należy odwoływać się w odwróconej kolejności, czyli odwołanie do grupy (\1) musi znaleźć się przed (z lewej strony) wyrażeniem definiującym tę grupę.

Z tego powodu z bardzo dużą ostrożnością trzeba stosować odwołania wsteczne wewnątrz cofnięć, szczególnie jeśli tworzymy bardziej skomplikowany wzorzec. Wielu programistów niemających doświadczenia z cofnięciami może źle zinterpretować sens wyrażenia regularnego. Jako ćwiczenie własne zostawiam Cię, Czytelniku, z ostatnim przykładem łączącym w sobie cofnięcia, wyprzedzenia oraz grupowanie z przechwytem:

```
'X_XaaaX Y_YbbbY'.match( /(?<=\1_(?<START>[A-Z]))\1[a-z]+(?=\k<START>)/g );
// ["aaa", "bbb"]
```

# Skorowidz

-, 79  
", 80  
\$, 67  
\$, znak, 20  
\*, znak, 20  
-, znak, 19, 28, 30  
^, znak, 27, 34, 67, 79  
|, znak, 25  
\\, znak, 26, 85  
\\0, 40  
\\b, 38, 40  
\\B, 40  
\\d, 31, 40  
\\D, 31, 40  
\\f, 40  
\\n, 40  
\\r, 40  
\\s, 40  
\\S, 40  
\\t, 40  
\\v, 40  
\\w, 40  
\\W, 40  
\\x00, 22, 38  
+, znak, 20

## A

alternatywa, 25  
  klasy znaków, 26  
  wielokrotna, 25  
analiza  
  adresu URL, 182  
  kodu pocztowego, 155  
  nazw i rozszerzeń plików, 193  
  numeru telefonu, 158

  plików INI, 173  
  zapisu czasu, 179  
  zapisu daty, 176  
analizowanie zawartości plików, 67  
antywzorzec, 29  
ASCII, 10  
  tablica znaków, 22

## B

białe znaki, 36  
Byte Order Mark, *Patrz* znacznik BOM

## C

ciągi znakowe  
  analiza bez użycia wyrażeń regularnych, 124  
  dynamiczne modyfikowanie, 7  
cofnięcie  
  negatywne, 225  
  pozytywne, 225

## D

dopasowanie  
  alternatywne, 119  
  cyfry, 28, 31  
  dosłowne, 19  
  dowolnego znaku, 19  
  globalne, 32, 61, 64  
  granice, 77  
  końca, 58  
  początku, 58  
  grupowanie, 41  
  z wykorzystaniem alternatywy, 44  
liter, 29, 33

dopasowanie  
 polskich znaków diakrytycznych, 29, 32  
 tabulatora  
 pionowego, 40  
 poziomego, 40  
 wsteczne, 39  
 wyrazu, 32  
 znaków  
 białych, 23  
 niedrukowanych, 23  
 specjalnych, 30  
 z rozszerzonego zakresu Unicode, 69  
 znaku  
 nowego wiersza, 40  
 NULL, 37, 40  
 powrotu karetki, 40  
 wysuwu wiersza, 40

**E**

ECMAScript 6  
 flaga u, 11

**F**

flaga, 60  
 case insensitive, 90, 94  
 definiowanie, 85  
 dotAll, 220  
 g, 38, 64, 95  
 global, 86, 87, 90, 98, 110  
 problemy, 95  
 i, 19, 61  
 m, 67  
 multiline, 136  
 sticky, 98, 111, 120  
 problemy, 95  
 u, 11, 23, 68, 69  
 unicode, 94  
 y, 73, 95  
 funkcja zwrotna, 116  
 argumenty, 117

**G**

granica  
 wyrazu, 38  
 dopasowania, 77

grupowanie dopasowań, 41  
 grupy  
 przechwytyjące, 102  
 analiza, 113  
 dopasowania alternatywne, 119  
 nazwane, 222  
 warunkowe, 102

**I**

index, 101  
 ISBN, 210

**K**

klasa znaków  
 negacja, 27  
 klasy znaków, 25  
 definiowanie, 26  
 dopasowywanie, 26  
 sekwencje negujące, 34  
 symbole specjalne, 79  
 zakres znaków, 28  
 znaki specjalne, 26  
 kodowanie znaków  
 ISO, 10  
 UTF, 10, 11  
 UTF-16, 13  
 różnice wobec UTF-8, 16  
 UTF-32, 11  
 UTF-8, 12  
 znacznik BOM, 15  
 kwantyfikator, 48, 78  
 dowolnej liczby powtórzeń, 108  
 niezachłanne, 53  
 przykłady, 51  
 skrótowe, 51  
 zachłanne, 53

**M**

metoda  
 charCodeAt, 131  
 codePointAt, 132  
 endsWith, 126  
 exec(), 99, 104  
 fromCharCode, 133



fromCodePoint, 133  
 includes, 126  
 indexOf(), 127  
 lastIndexOf, 128  
 Number.prototype.toString, 18  
 RegExp.prototype.exec, 97  
   flaga global, 98  
   flaga sticky, 98  
 RegExp.prototype.test, 17, 18, 25, 53, 73, 87, 94  
 RegExp.prototype.toString, 91  
 RegExp.test, 19  
 search(), 107  
   flaga global, 109  
   flaga sticky, 109  
 startsWith, 125  
 String.prototype.includes, 19  
 String.prototype.indexOf, 19  
 String.prototype.match, 53, 80, 86, 110  
   flaga global, 110  
   flaga sticky, 111  
 String.prototype.replace, 32, 41, 86, 114, 142  
   argumenty, 117  
   zmodyfikowana, 224  
 String.prototype.search, 106  
 String.prototype.split, 121  
 String.prototype.toLowerCase, 20  
 String.prototype.trim, 136  
 toString, 19  
 modyfikacja wskazanych wyrazów, 167  
 modyfikator, *Patrz* flaga

## N

negative lookahead, *Patrz* wyprzedzenia  
 negatywne  
 NULL, 37

## O

odwołania wsteczne, 41, 224

## P

positive lookahead, *Patrz* wyprzedzenia  
 pozytywne  
 predefiniowane  
   sekwencje znakowe, 40

znaki, 31  
 przechwycenie separatora, 122

## R

RegExp  
   dostęp do właściwości obiektu, 114  
   konstruktor, 84  
   sekwencja ucieczki, 84  
   obsługa flag, 90  
   source, 91  
   właściwości obiektu, 90, 91  
 REGON, 208

## S

sekwencja ucieczki, 20, 22, 84  
   stosowanie, 76  
 sprawdzanie podzielności  
   przez 2, 151  
   przez 4, 151  
   przez 5, 151  
 sprawdzenie liczby  
   dwójkowej, 145  
   szesnastkowej, 145  
   zawiera się w zakresie, 153  
 symbole specjalne, 79, 81

## T

tryb  
   ignorowania białych znaków, 25  
   ignorowania wielkości znaków, 61

## U

Unicode, 10, 11, 19, 69, 129, 227  
   grupy znakowe, 227  
   rozszerzony zakres, 69  
   tablica znaków, 10, 22  
   tworzenie ciągów znakowych, 130  
   zakres znaków, 29  
 usuwanie  
   białych znaków, 135  
   komentarzy, 171  
   przedrostków, 160  
   znaczników HTML, 164

**W**

walidacja  
danych, 9, 32  
adresu  
  e-mail, 195  
  IP, 199  
  MAC, 17  
liczby  
  całkowitej, 138  
  w systemie wykładniczym, 142  
  z częścią ułamkową, 141  
numeru karty kredytowej, 185  
poprawności MAC Address, 201  
weryfikacja  
  europejskiego numeru płatnika VAT, 214  
  kolorów  
    HEX, 148  
    HSL, 148  
    RGB, 148  
  numeru  
    dowodu osobistego, 204  
    ISBN, 210  
    NIP, 206  
    PESEL, 202  
    REGON, 208  
  poprawności hasła, 180  
  zapisu liczb rzymskich, 188  
wykrywanie powtórzeń, 168  
właściwość  
  index, 101  
  lastIndex, 73, 99, 101, 109  
  RegExp.flags, 61  
  RegExp.input, 92  
  RegExp.lastMatch, 92  
  RegExp.lastParen, 92  
  RegExp.leftContext, 92  
  RegExp.prototype.flags, 90  
    kolejność flag, 90  
  RegExp.rightContext, 92

wyprzedzenia  
  negatywne, 56  
  pozytywne, 56  
wyrażenia regularne  
  definicja, 7  
  jako separator, 121  
  przykłady, 135  
  tworzenie, 83  
wzorzec regexp  
  definiowanie flag, 85  
  komentowanie, 88  
  tworzenie, 86

**Z**

zakresy znaków, 31  
  predefiniowane, 40  
zamiana polskich znaków diakrytycznych na  
litery łańskie, 163  
zastępowanie znaczników HTML, 165  
znacznik  
  BOM, 15  
znaki z określonego zakresu, 136  
znaki specjalne, 20, 26, 195  
  ", 22, 26  
  \$, 20  
  \*, 20  
  ., 18  
  ^, 27  
  +, 20

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

# JavaScript

## Świadomie stosuj wzorce – to ułatwi Ci pracę!

Wyrażenia regularne w JavaScriptcie, podobnie jak we wszystkich innych językach programowania, wymyślono po to, żeby ułatwić programistom pracę. Są niezastąpione chociażby przy walidacji informacji wprowadzanych przez użytkowników, przeszukiwaniu zbiorów danych czy automatyzacji wielu zadań. Niestety, duża część programistów — i to nie tylko początkujących — omija wyrażenia regularne szerokim łukiem w przekonaniu, że zbyt trudno je zrozumieć, a pomyłka może sporo kosztować. Z tej książki dowiesz się, jak należy czytać oraz samodzielnie konstruować i testować wyrażenia regularne, żeby służyły Twoim celom. Szybko przekonasz się, jak wygodne jest to rozwiązanie.

Niezależnie od tego, czy jesteś nowicjuszem, czy programujesz od lat, znajdziesz tu coś dla siebie. Poznasz metody i zasady pracy z regexami, nauczysz się testować wzorce i dopasowywać je do swoich zamierzeń. Dogłębna analiza składni wyrażeń regularnych oraz ich zastosowania została tu poszerzona o omówienie potencjalnych problemów i częstych błędów popełnianych przez osoby stawiające pierwsze kroki w tym obszarze. Przeczytaj, wypróbuj wyrażenia regularne i zacznij używać ich na co dzień, a Twoja praca w JavaScriptcie stanie się znacznie bardziej efektywna!

- Na początek nieco teorii
- Składnia wyrażeń regularnych w JavaScriptcie
- Praca z wyrażeniami regularnymi w JavaScriptcie
- Przykładowe wyrażenia regularne z omówieniem
- Przyszłość wyrażeń regularnych w JavaScriptcie

## Regexy – używaj ich regularnie!

	<p><i>Sprawdź nasze szkolenia!</i></p> <p>SZKOLENIA</p>  <p>AKADEMIA IT &amp; BUSINESS</p> <p>WWW.SZKOLENIA.HELION.PL</p>	<p><b>KOD KORZYŚCI</b> <i>Sięgnij po więcej!</i></p> 
 <b>helion.pl</b>		
 <b>HELION SA</b> ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 hellon@helion.pl		
<b>INFORMATYKA W NAJLEPSZYM WYDANIU</b>		<p>ISBN 978-83-283-3892-0</p>  <p>9 788328 338920</p> <p>Cena: 49,00 zł</p>