Tomasz Sochacki

JavaScript

Tworzenie nowoczesnych aplikacji webowych

Helion

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

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

ISBN: 978-83-283-5637-5

Printed in Poland.

Spis treści

Rozdział 5.

Operacje na ciągach znakowych

Omówiłem dotychczas najważniejsze elementy składni języka JavaScript, pora zatem na rozpoczęcie używania ich w praktyce. Na początek przyjrzymy się operacjom na ciągach znakowych. Z wartościami typu `string` spotkasz się praktycznie w każdym projekcie, dlatego warto dobrze poznać możliwości, jakie daje nam język JavaScript do ich obsługi.

Tworzenie ciągów znakowych

W poprzednich rozdziałach używaliśmy trzech sposobów na tworzenie ciągów znakowych:

```
const name = 'Tomek';
const name = "Tomek";
const name = `Tomek`;
```

Pierwsze dwa zapisy są sobie równoważne. W języku JavaScript nie ma znaczenia czy zastosujemy apostrofy i cudzysłowy. Nieco inaczej jest w trzecim przykładzie, gdzie zastosowaliśmy tzw. *template strings*. Zapis ten pozwala na interpolowanie instrukcji wewnątrz tzw. „odwróconych apostrofów".

Zobaczmy jak w praktyce można zastosować *template strings*:

```
const name = 'Jan';
const text_1 = 'Mam na imię: ' + name;
const text_2 = `Mam na imię: ${name}`;

text_1; // "Mam na imię: Jan"
text_2; // "Mam na imię: Jan"
text_1 === text_2; // true
```

Na początku stworzyliśmy stałą przechowującą ciąg znakowy Jan. Następnie w stałej `text_1` zastosowaliśmy tzw. konkatenację ciągów znakowych, przeciążając operator +. Taki zapis był najczęściej stosowaną metodą łączenia ciągów znakowych. Istnieje jeszcze sposób z użyciem metody `concat`, jednakże jest on rzadko spotykany i osobiście nie miałem nigdy potrzeby użyć go w praktyce.

Zauważ jednak, w jaki sposób stworzyliśmy wartość stałej `text_2`. Tutaj właśnie użyliśmy składni *template strings* z wykorzystaniem interpolacji innego wyrażenia, a dokładniej stałej

name. Aby użyć innej zmiennej czy funkcji, w *template strings* należy zastosować składnię ze znakiem dolara i nawiasami klamrowymi:

```
`jakiś tekst ${someVariable}`;
`jakiś tekst ${someFunction()}`;
```

Możemy tutaj używać praktycznie każdej poprawnej instrukcji w JavaScript, np. operatora trójargumentowego:

```
`jakiś tekst ${ someNumber > 1 ? valueForTrue : valueForFalse}`;
```

Co więcej, *template string* może zawierać inne, wewnętrzne *template strings*, jednakże zalecam unikać takich konstrukcji, gdyż są one trudne w analizie i zdarza się, że niektóre edytory niepoprawnie interpretują takie zapisy, co nieco utrudnia tworzenie kodu. Jeśli zaszłaby konieczność zagnieżdżania ciągów generowanych przez *template strings*, to najczęściej lepszym rozwiązaniem będzie wygenerowanie osobno kolejnych ciągów i ich końcowe złączenie.

Template strings to składnia wprowadzona w wersji ECMAScript 6 i jest obsługiwana praktycznie przez wszystkie nowoczesne przeglądarki. Nie należy jednak jej nadużywać; gdy nie potrzebujemy przetwarzać żadnych dodatkowych warunków czy używać konkatenacji, lepiej stworzyć ciąg znakowy, wykorzystując w tym celu metodę z cudzysłowami lub apostrofami (tutaj wybór zależy już od indywidualnych preferencji).

Istnieje jeszcze jeden sposób na określenie tego, jakie znaki mają znaleźć się w ciągu. Mowa tutaj o zapisie tzw. punktów kodowych znaków zgodnie z tablicą ASCII lub tablicą Unicode. Jeśli spojrzymy do tablicy ASCII, to zobaczymy, że każdy znak posiada swój indywidualny punkt kodowy w zapisie szesnastkowym. Na przykład mała litera a posiada punkt kodowy 61:

```
'a' === '\x61';   // true
'a' === '\u0061'; // true
```

Zwróć uwagę na dwa sposoby zapisu punktu kodowego dla małej litery a. Po znaku \ znajduje się litera x lub u, a po niej 2 lub 4 znaki określające numer kodowy w zapisie szesnastkowym. Generalnie tablica Unicode stanowi zbiór wszystkich znaków jakie są stosowane w praktyce, w tym m.in. zawiera litery różnych alfabetów, znaki specjalne, symbole itp. Tablica ASCII jest zawarta w tablicy Unicode i stanowi jej początkowe elementy.

Ważne jest, aby pamiętać, że skróconą formę z użyciem \x można stosować tylko dla znaków o punktach kodowych do FF (czyli maksymalnie dla pierwszych 255 znaków tablicy Unicode). Gdy punkt kodowy jest zapisany przy użyciu 3 lub 4 znaków, konieczne jest użycie zapisu \u. W tym wypadku należy jednak pilnować, aby zawsze podawać dokładnie 4 znaki, w razie potrzeby dopisując zera na początku.

Bardziej szczegółowo temat punktów kodowych i sposobów obsługi znaków w języku JavaScript został omówiony w książce *JavaScript. Wyrażenia regularne dla programistów*.

Modyfikacje wielkości znaków

Jedną z często wykonywanych operacji jest zmiana wielkości znaków w ciągu. Jest to przydatne na przykład w celu porównania wartości pochodzącej od użytkownika (np. z pola w formularzu na stronie internetowej) z jakimś ciągiem referencyjnym. Dostępne są dwie proste metody służące do modyfikacji wielkości znaków:

```
const name = 'Jan Kowalski';

name.toLowerCase(); // "jan kowalski"
name.toUpperCase(); // "JAN KOWALSKI"
```

Metody `toLowerCase` i `toUpperCase` modyfikują zasadniczo tylko znaki, które są literami alfabetu w różnych językach. Nie modyfikują natomiast w żaden sposób cyfr czy innych znaków nie będących literami:

```
'123'.toLowerCase(); // "123"
'!!!'.toLowerCase(); // "!!!"
```

Metody działają poprawnie praktycznie z większością popularnych znaków, w tym z literami ze znakami diakrytycznymi:

```
'ą'.toUpperCase(); // "Ą"
'Ą'.toLowerCase(); // "ą"
```

Zdarzają się co prawda pewne wyjątki, w których te metody mogą zachowywać się nie do końca poprawnie, ale nie będę się nimi zajmować w poradniku przeznaczonym dla osób zaczynających dopiero naukę języka JavaScript. Osoby zainteresowane tym zagadnieniem zachęcam do analizy dokumentacji dla metod `toLocaleLowerCase` oraz `toLocaleUpperCase`.

Zobaczmy jeszcze, w jaki sposób można wykorzystać te metody do modyfikacji wielkości znaków w łączonych ciągach znakowych:

```
const firstName = 'Jan';
const lastName = 'Kowalski'

const name1 = (firstName + lastName).toLowerCase();     // "jan kowalski"
const name = `${firstName} ${lastName}`.toLowerCase(); // "jan kowalski"
```

Jest to dobry przykład, dokładnie pokazujący, co tak naprawdę robi składnia z użyciem tzw. odwróconych apostrofów. Metodę `toLowerCase` możemy wywołać jedynie na obiekcie typu `string`. Dlatego w pierwszym przykładzie musieliśmy połączenie ciągów otoczyć nawiasem, aby metoda została wywołana na ciągu będącym konkatenacją `firstName` i `lastName`.

Gdy używamy odwróconych apostrofów, nawiasy nie są potrzebne, ponieważ składnia ta zapewnia, że zawsze wynikowo dostaniemy jeden ciąg znakowy typu `string` i to na nim wywołamy naszą metodę `toLowerCase`. Jest to bezpieczny sposób, ponieważ zapis `` `...` `` gwarantuje, że wygenerowana wartość zawsze będzie stringiem, w ostateczności pustym ciągiem znakowym, ale nadal typu `string`. Pozwala nam to stosować metody `String.prototype` bez obaw, że moglibyśmy próbować użyć jej na obiekcie niebędącym stringiem.

Wycinanie fragmentu ciągu

Często zdarzy się, że będziemy potrzebowali wyciągnąć jakiś fragment z dłuższego ciągu znakowego. Możemy to osiągnąć na kilka sposobów. Na początek jednak wróćmy do jednej z istotnych, podstawowych kwestii dotyczących wartości typu `string`.

Otóż wszystkie ciągi znakowe w JavaScript są iterowalne, co oznacza (w pewnym uproszczeniu) że możemy iterować po każdym pojedynczym znaku. W rozdziale tym spróbujemy z ciągu:

```
const str = 'Jakiś przykładowy tekst.';
```

wydobyć fragment:

```
'przykładowy'
```

i przypisać go do nowej zmiennej. Istnieje możliwość dostępu do każdego pojedynczego znaku w ciągu przy użyciu zapisów:

```
str[0]; // "J"
str.charAt(0); // "J"
```

Metoda `charAt` jest raczej rzadziej stosowana, przyjmuje ona jako parametr indeks, czyli pozycję znaku, który chcemy pobrać. Pamiętaj, że podobnie jak w przypadku tablic, pierwszy znak zawsze znajduje się pod indeksem zerowym. Pierwszy zapis mógłby sugerować, że zmienna `str` jest tablicą, jednakże nie jest to prawda. Zapis ten po prostu pozwala pobierać poszczególne znaki przy użyciu składni podobnej do tablic, nie wywołamy jednak na ciągu znakowym metod znajdujących się w `Array.prototype`.

Skoro wiemy już, że możemy iterować po ciągu, wyodrębniając jego kolejne elementy, to spróbujmy wykorzystać w praktyce znaną nam już pętlę `for`:

```
const str = 'Jakiś przykładowy tekst.';
let fragment = '';

for (let i = 0; i < str.length; i++) {
  if (i >= 6 && i < 17) {
    fragment += str[i];
  }
  if (i >= 17) {
    break;
  }
}

fragment; // "przykładowy"
```

Jak widać, udało nam się osiągnąć zamierzony efekt. Przeanalizujmy kolejno poszczególne etapy. Najpierw zadeklarowaliśmy nasz ciąg `str` oraz drugi, o nazwie `fragment`. Zwróć uwagę na to, że `fragment` jest zadeklarowany przy użyciu słowa `let`, co pozwala nam modyfikować jego wartość po zadeklarowaniu. Jest to istotne, gdyż za chwilę będziemy dodawać do niego kolejne znaki.

Następnie iterujemy po tablicy i w każdej iteracji sprawdzamy, czy indeks zawiera się między 6 a 16, co odpowiada pozycjom szukanych przez nas znaków. Jednocześnie ustawiamy warunek, aby po odnalezieniu ostatniego interesującego nas znaku pętla zakończyła działanie, co zapewnia nam instrukcja `break`.

Przyjrzyj się jeszcze poleceniu:

```
fragment += str[i];
```

jest ono tożsame z zapisem:

```
fragment = fragment + str[i];
```

Zgodzisz się jednak zapewne ze mną, że nie jest to najwygodniejsza forma pobierania fragmentu ciągu znakowego, szczególnie gdy w aplikacji musimy wielokrotnie wykonywać takie operacje. Fakt, że będziemy wielokrotnie używać tego samego kodu mówi nam, że należałoby taką operację wydzielić do osobnej funkcji. Spróbujmy więc zrobić tak:

```
function getFragment(value, startIndex, endIndex) {
  const start = startIndex || 0;
  const end = endIndex || value.length;
  let fragment = '';

  for (let i = start; i < end; i++) {
    if (i >= start && i < end) {
      fragment += value[i];
    }
    if (i >= end) {
      break;
    }
  }
  return fragment;
}

getFragment('Jakiś przykładowy tekst.', 6, 17); // "przykładowy"
```

Oczywiście wartość zwracaną przez naszą funkcję możemy od razu przypisać do nowej zmiennej:

```
const newStr = getFragment('Jakiś przykładowy tekst.', 6, 17);
newStr; // "przykładowy"
```

Nasza funkcja pobiera trzy parametry — ciąg, który chcemy analizować, indeks pierwszego elementu, jaki chcemy wyciąć, oraz indeks elementu, na którym mamy zakończyć analizę, lecz bez wyciągania tego ostatniego znaku. Zauważ, że nasza pętla kończy wycinanie fragmentu, gdy indeks jest równy 17. W kolejnej iteracji warunek 17 < 17 daje `false` i pętla kończy działanie, a dokładniej mówiąc wchodzi w blok drugiej instrukcji `if` i wywołuje polecenie `break`. Za chwilę dowiesz się, dlaczego zastosowaliśmy właśnie taką formę podawania indeksów zamiast wskazywać jako `endIndex` ostatni interesujący nas znak.

Przypomnij sobie teraz podrozdział omawiający dziedziczenie w JavaScript i zagadnienie tzw. prototypów. Wiesz już, że ciągi znakowe dziedziczą po `String.prototype`, gdzie znajduje się m.in. metoda `toLowerCase`, co pozwala zastosować ją bezpośrednio na wartości typu `string`.

Spróbujmy więc dodać naszą funkcję do ogólnego prototypu `String.prototype`, gdyż w zasadzie jest ona na tyle uniwersalna, że możliwe byłoby zastanowienie się nad taką operacją:

```
if (typeof String.prototype.getFragment === 'undefined') {
  String.prototype.getFragment = function (startIndex, endIndex) {
    const start = startIndex || 0;
    const end = endIndex || this.length;
    let fragment = '';

    for (let i = start; i < end; i++) {
      if (i >= start && i < end) {
        fragment += this[i];
      }
      if (i >= end) {
        break;
      }
    }
    return fragment;
  };
}

'Jakiś przykładowy tekst.'.getFragment(6, 17); // "przykładowy"
```

Zwróć uwagę na pierwszą linijkę:

```
if (typeof String.prototype.getFragment === 'undefined') {
```

Pamiętaj, że gdy kiedykolwiek będziesz chciał dodać coś do prototypu (co nie jest ogólnie dobrą praktyką, ale czasami bywa przydatne), powinieneś upewnić się najpierw, że metody takiej nie ma w danym prototypie. Zawsze należy uważać, gdyż nadpisanie metody z prototypu mogłoby uszkodzić inne fragmenty kodu, np. biblioteki zewnętrzne wykorzystujące metodę i oczekujące wersji pierwotnej, a nie naszej, nadpisującej natywną metodę. Z drugiej strony gdy zaimplementujemy w ten sposób metodę `String.prototype.getFragment`, narażamy się na niebezpieczeństwo, gdy np. w kolejnej wersji ECMAScript zostanie dodana natywna metoda o takiej samej nazwie, lecz zachowująca się nieco inaczej niż nasza implementacja.

Dodatkowo w tym momencie nie mamy już parametru `value`, gdyż metoda jest wywoływana bezpośrednio na ciągu znakowym i jest on dostępny jako `this`.

Przeszliśmy długą drogę, ale ostatecznie uzyskaliśmy metodę, która może być wywołana bezpośrednio na obiekcie `string`. Jak widać, niesie to jednak za sobą pewne ryzyko, którego musimy być świadomi, szczególnie w większych aplikacjach, gdzie nie mamy pełnej kontroli nad całym kodem, który tworzony jest przez wielu programistów. W takich sytuacjach modyfikacje prototypów najczęściej nie są dobrą praktyką.

W tym momencie doszliśmy do sytuacji, w której co prawda mamy funkcjonalność działającą zgodnie z naszymi oczekiwaniami, jednakże wymagało to dość sporej ilości kodu i wykonania ryzykownej operacji ingerowania w prototyp. Cofnijmy się więc o kilka kroków i zacznijmy od początku…

Co zrobić, aby wyciągnąć jakiś określony fragment z ciągu znakowego? Na przykład użyć dostępnej już metody `String.prototype.slice`:

```
'Jakiś przykładowy tekst.'.slice(6, 17); // "przykładowy"
```

Jak widać, metoda ta robi dokładnie to samo co nasza funkcja `getFragment`. Teraz wiesz już, dlaczego wcześniej zastosowaliśmy właśnie taką implementację dla `startIndex` i `endIndex`. Mógłbyś w tym miejscu zapytać, po co właściwie tworzyliśmy własną metodę, skoro dysponujemy gotowymi rozwiązaniami?

Otóż zrobiłem to celowo. Wiele kursów i poradników wymienia wiele metod dostępnych w różnych wersjach JavaScript i jak najbardziej warto je znać i stosować. Warto jednak również pamiętać, że praktycznie wszystkie popularnie stosowane metody `String.prototype`, `Array.prototype` itp. można by zastąpić własnymi implementacjami, które robiłyby dokładnie to samo. Jest to co prawda pewne uproszczenie, ale nie wchodząc w szczegóły działania silnika JavaScript i mechanizmów optymalizacyjnych można przyjąć, że dla naszych potrzeb implementacje takie mogłyby być równoważne z metodami natywnymi.

Szczególnie na początku nauki języka JavaScript gorąco zachęcam Cię do takich zabaw i prób tworzenia własnych implementacji zachowujących się tak samo jak metody natywne. Jako ćwiczenie możesz na przykład spróbować zaimplementować funkcję `convertToLowerCase`, która zachowa się jak metoda `String.prototype.toLowerCase` w zakresie podstawowego alfabetu łacińskiego (czyli np. bez polskich znaków diakrytycznych). Przypomnij sobie w tym momencie, co mówiłem o możliwości zapisywania znaków poprzez ich punkty kodowe Unicode i spróbuj dostrzec zależność w ułożeniu punktów kodowych dla małych i wielkich liter alfabetu łacińskiego.

Omówmy zatem nieco dokładniej metodę `String.prototype.slice`, gdyż w rzeczywistości daje ona znacznie większe możliwości niż nasza metoda `getFragment`. Generalnie metoda przyjmuje dwa parametry, `index` początkowy i końcowy, przy czym są to parametry opcjonalne. Jeśli jednak chcemy określić indeks końcowy, konieczne jest też wskazanie indeksu początkowego (na przykład jako wartość 0 gdy zaczynamy wycinać od początku ciągu). Przeanalizujmy kilka przykładów:

```
'abcde'.slice(0, 2);  // "ab"
'abcde'.slice(0, 5);  // "abcde"
'abcde'.slice(0, 10); // "abcde"
'abcde'.slice(2, 4);  // "cd"
'abcde'.slice(2, 10); // "cde"
```

Zauważ, że po podaniu jako `endIndex` wartości większej niż dostępna długość ciągu nie zostaje zgłoszony żaden błąd i po prostu dostajemy ciąg od wskazanego `startIndex` do samego końca.

Co ciekawe, indeksy te mogą być również wartościami ujemnymi. Jak wtedy zachowuje się metoda `slice`?

```
'abcde'.slice(-2);  // "de"
'abcde'.slice(-5);  // "abcde"
'abcde'.slice(-10); // "abcde"
```

Gdy podamy tylko `startIndex` jako wartość ujemną, pobierzemy wskazaną ilość znaków licząc od końca ciągu. Jest to przydatna i dość często spotykana w aplikacjach funkcjonalność. Warto poćwiczyć działanie tej metody na różnych przykładach (zachęcam Cię do otworzenia w tym momencie konsoli narzędzi deweloperskich w przeglądarce i podjęcia własnych prób):

```
'abcde'.slice(-3, -1); // "cd"
'abcde'.slice(-4, -2); // "bc"
'abcde'.slice(-4, -1); // "bcd"
```

Pierwszy przykład w luźnym tłumaczeniu oznacza: wyciągnij trzy znaki licząc od końca (`-3`), ale bez ostatniego (`-1`). Analogicznie drugi przykład oznacza: pobierz cztery ostatnie znaki (`-4`), lecz bez dwóch na samym końcu (`-2`). Stosując jednak tego typu — nieco bardziej złożone — wyciąganie fragmentów, należy bardzo dokładnie weryfikować, czy na pewno uzyskany ciąg znakowy jest tym, czego się spodziewamy. Metoda `slice` wywołana bez żadnych parametrów przyjmie wartości domyślne, co spowoduje po prostu zwrócenie całego ciągu wejściowego. Pozwala nam to np. na stworzenie kopii ciągu bez ingerowania w zmienną przechowującą oryginalny ciąg znakowy.

Sprawdzanie początku i końca ciągu znakowego

Częstą operacją jest również przeszukiwanie ciągów znakowych, przy czym możemy wyróżnić tutaj dwa podstawowe rodzaje takiego działania — analizę początku lub końca ciągu oraz wyszukiwanie jakieś frazy wewnątrz całego ciągu znakowego. Najpierw zajmiemy się pierwszym przypadkiem i omówimy kilka sposobów na analizę znaków znajdujących się na początku lub na końcu wartości typu `string`.

Załóżmy, że zmienna `url` przechowuje jakiś adres strony internetowej. Naszym zadaniem jest sprawdzenie, czy adres ten zaczyna się od protokołu `http`. Możemy w tym celu wykorzystać metodę `String.prototype.startsWith`:

```
const url = 'http://google.com';
url.startsWith('http'); // true
url.startsWith('HTTP'); // false
```

Metoda `startsWith` dała pozytywny wynik dla ciągu `http`; zwróć jednak uwagę na istotny szczegół. Otóż metoda ta bierze pod uwagę wielkość znaków, czyli ciąg `http` nie jest dla niej równy ciągowi HTTP. Można sobie poradzić z tym problemem poprzez wcześniejsze przekonwertowanie wszystkich wielkich liter na małe:

```
url.toLowerCase().startsWith('http'); // true
```

Należy o tym pamiętać, jeśli chcemy sprawdzić początek ciągu bez względu na wielkość liter. Metoda `startsWith` przyjmuje jeszcze jeden parametr i jest nim liczba, wskazująca na `index` znaku, od którego ma się rozpocząć wyszukiwanie. Żaden ze znaków przed wskazanym indeksem nie jest brany pod uwagę:

```
url.startsWith('google'); // false
url.startsWith('google', 7); // true
```

Znak o indeksie `7` to litera `g`, w związku z tym w powyższym przykładzie do analizy został wzięty tylko fragment `google.com`, a początek `http://` został zignorowany. Zanim w JavaScript została wprowadzona metoda `startsWith`, często spotykanym rozwiązaniem było użycie metody `slice` aby wyciąć fragment ciągu znakowego, po czym wycięty fragment przyrównano do innego ciągu:

```
url.slice(0, 4) === 'http'; // true
```

Alternatywą dla użycia metody `slice` i `startsWith` jest zastosowanie tzw. wyrażeń regularnych, które pozwalają w prosty sposób tworzyć bardziej elastyczne warunki, ignorować wielkość znaków itp. Temat wyrażeń regularnych (tzw. *regexp*) nie będzie tu szeroko omawiany, gdyż są to zagadnienia na poziomie bardziej zaawansowanym. Osoby zainteresowane tym zagadnieniem zachęcam do lektury książki *JavaScript. Wyrażenia regularne dla programistów*. Poniżej znajduje się przykładowe wyrażenie regularne, wyszukujące ciąg `http` na początku stringa:

```
/^http/i.test(url); // true
```

W języku JavaScript dysponujemy również metodą `endsWith`, która — jak wskazuje jej nazwa — pozwala sprawdzić, czy ciąg znakowy kończy się określonym fragmentem:

```
url.endsWith('com'); // true
```

Jeśli nie zależy nam na wielkości znaków, to podobnie jak we wcześniejszych przykładach, można wykorzystać dodatkową metodę `toLowerCase`, aby wymusić przekonwertowanie ciągu na małe litery:

```
url.toLowerCase().endsWith('com'); // true
```

Metoda `endsWith` przyjmuje również drugi parametr, który określa indeks znaku, od którego należy zignorować dalszą część ciągu. Zobaczmy to na przykładach:

```
const url = 'http://google.com';

url.endsWith('.', 14); // true
url[14]; // "c"
```

Znak o indeksie `14` to litera `c` (pamiętaj, że indeksy liczone są od 0), dlatego w powyższym przykładzie, w pewnym uproszczeniu można przyjąć, że metoda odcięła znaki o indeksie `14` i większych, czyli zignorowała fragment `com` i jako ciąg do analizy przyjęła `http://google.`. Ciąg taki kończy się znakiem kropki, stąd otrzymujemy — zgodnie z oczekiwaniami — wynik `true`.

Przeanalizujmy jeszcze inny przykład:

```
url.endsWith('google', 13); // true
url[13]; // "."
```

W tym wypadku sprawdzamy, czy ciąg złożony z pierwszych trzynastu znaków, czyli `http://google` (bez kropki) posiada na końcu fragment `google`. Ponownie metoda `endsWith` dokonała odpowiedniego odcięcia znaków o indeksie 13 i wyższych i zwróciła wynik `true`.

Jednocześnie należy pamiętać, że metody `startsWith` oraz `endsWith` nie modyfikują w żaden sposób oryginalnych ciągów, a zatem przekazanie im drugiego parametru w celu bardziej precyzyjnej analizy nie wpłynie na wartość przechowywaną w zmiennej `url`.

Przeszukiwanie ciągu znakowego

W praktyce najczęściej nie interesuje nas początek lub koniec ciągu, lecz chcemy dowiedzieć się, czy cały ciąg znakowy zawiera jakiś określony fragment. W języku JavaScript możemy takie zadanie rozwiązać na kilka sposobów. Załóżmy, że mamy ciąg:

```
const str = 'Jakiś przykładowy tekst.';
```

i chcemy sprawdzić, czy znajduje się w nim ciąg `przykładowy`. Przeanalizujmy różne sposoby na wykonanie takiej analizy.

Metoda includes

Najwygodniejszą i obecnie jedną z najczęściej stosowanych we wspomnianym wyżej celu metod jest `String.prototype.includes`. Metoda ta przyjmuje jako parametr szukany ciąg znakowy i zwraca wartość `true` lub `false`, w zależności od tego, czy uda się pozytywnie wyszukać wskazany ciąg.

```
str.includes('przykładowy'); // true
```

Dodatkowo metoda `includes` może przyjąć jeszcze drugi, opcjonalny parametr, który wskazuje indeks, od którego ma zostać rozpoczęte wyszukiwanie:

```
str.includes('przykładowy', 6); // false
```

W powyższym przykładzie metoda `includes` zignorowała pierwsze 6 znaków i do analizy wzięła tylko ciąg zaczynający się od znaku `x`. Podobny efekt można by uzyskać, gdybyśmy najpierw odcięli te znaki metodą `slice` i dokonali wyszukania na takim podciągu:

```
str.slice(7); // "rzykładowy tekst."
str.slice(7).includes('przykładowy'); // false
```

Metoda zwraca `false`, ponieważ chcemy wyszukać całe słowo `przykładowy`, a w skróconym ciągu znajduje się tylko jego fragment, bez litery `p`. Z metodą tą związany jest jednak pewien problem, analogiczny jak w metodach `startsWith` oraz `endsWith` — nie mamy możliwości

zignorowania wielkości znaków. Jeśli interesuje nas wyszukanie słowa `przykładowy` zapisanego z małych liter, wielkich lub ich dowolnej kombinacji, to również tym razem mamy dwa rozwiązania. Pierwsze polega na jawnym wywołaniu metody np. `toLowerCase` przed metodą `includes`, a drugie na użyciu wyrażeń regularnych. W tym przypadku jednak stworzenie dobrego *regexp* nie jest takie proste i wymaga uwzględnienia wielu opcji, dlatego nie będę się tym zajmować w niniejszej książce.

Wyszukiwanie metodą indexOf

Przed pojawieniem się metody `String.prototype.includes` popularnym sposobem wyszukiwania w ciągach znakowych było użycie metody `String.prototype.indexOf`. Nie należy jednak metod tych traktować jako bezpośrednich alternatyw, gdyż zachowują się one nieco inaczej. Otóż metoda `indexOf` pobiera jako pierwszy (obowiązkowy) parametr ciąg, który chcemy wyszukać. Zwraca jednak nie wartość `boolean`, a indeks, pod jakim znaleziono pierwszy znak szukanego ciągu. Zobaczmy to na naszym przykładzie:

```
str.indexOf('przykładowy'); // 6
str[6]; // "p"
```

Zwróć uwagę na to, że otrzymaliśmy wartość typu `number`, czyli indeks pierwszego znalezionego znaku — litery e. We wcześniejszych rozdziałach mówiliśmy o możliwości jawnego przekonwertowania dowolnej wartości na typ `boolean` przy użyciu dwóch znaków wykrzyknika. Spróbujmy więc wykorzystać tę metodę, aby otrzymać wartość `true`/`false` analogicznie jak w metodzie `includes`:

```
!!str.indexOf('przykładowy'); // true
```

Liczba 5 — jak pamiętamy — jest konwertowana do `true`. Teoretycznie można by więc uznać, że metoda `indexOf` mogłaby być użyta wewnątrz instrukcji warunkowej `if`, na przykład w formie:

```
if (str.indexOf('przykładowy')) {
  // instrukcje dla true
}
```

W praktyce jednak jest to bardzo ryzykowne rozwiązanie i wcześniej czy później może doprowadzić do poważnych błędów w działaniu aplikacji. Nie wspominałem do tej pory co się dzieje, gdy nie uda się znaleźć szukanego fragmentu w całym ciągu znakowym. Otóż w takiej sytuacji metoda `indexOf` zwraca wartość `-1` (minus jeden):

```
str.indexOf('xyz'); // -1
```

Początkowo mogłoby się to wydawać nieco dziwne — dlaczego w zasadzie nie zwrócić po prostu wartości zerowej, która np. w instrukcji warunkowej `if` zostałaby przekonwertowana do `false`? Otóż takie rozwiązanie miałoby poważną wadę — nie pozwoliłoby na wyszukanie fragmentu, który znajdowałby się na samym początku ciągu:

```
str.indexOf('Jakiś');   // 0
!!str.indexOf('Jakiś'); // false
```

Z tych właśnie powodów pamiętaj, aby w przypadku metody `indexOf` zawsze odnosić jej wynik do wartości `-1` lub do wartości większej lub równej 0:

```
if (str.indexOf('przykładowy') !== -1) {
  // instrukcje dla true
}
// lub
if (str.indexOf('przykładowy') >= 0) {
  // instrukcje dla true
}
```

Najczęściej jednak spotyka się zapis sprawdzający, czy uzyskany wynik nie jest równy `-1`.

Metoda `indexOf` przyjmuje również drugi parametr, wskazujący na indeks znaku, od którego ma się rozpocząć wyszukanie:

```
str.indexOf('przykładowy', 6); // 6
str[6] // "p"

str.indexOf('przykładowy', 7); // -1, nie znaleziono, ponieważ analizowany ciąg to:
str.slice(7); // "rzykładowy tekst."

str.indexOf('przykładowy', 1); // 6, analizowany ciąg to:
str.slice(1) // "akiś przykładowy tekst."
```

Drugi parametr w pewnym uproszczeniu odcina wskazaną ilość początkowych znaków, jednakże zwróć uwagę, że w przypadku, gdy obcięty ciąg zawiera szukany fragment, metoda zwraca indeks pierwszego znalezionego znaku zawsze w odniesieniu do całego ciągu. Dlatego w razie wywołania metody z drugim parametrem równym `1` nie otrzymaliśmy zwrotnie indeksu równego `4`, lecz `5`, co stanowi indeks litery `e` w całym ciągu `str`. Jest to istotne jeśli używamy metody `indexOf` w celu nie tyle sprawdzenia istnienia danego fragmentu, lecz w celu sprawdzenia jego pozycji w całym ciągu. Jeśli w takiej sytuacji chcielibyśmy faktycznie zignorować pierwszy znak, to konieczne byłoby wcześniejsze wywołanie metody `slice`:

```
str.indexOf('przykładowy', 1);  // 6
str.slice(1).indexOf('przykładowy', 1); // 5
```

Metoda lastIndexOf do analizy ciągów znakowych

Metoda `indexOf` próbuje znaleźć pierwsze wystąpienie wskazanego ciągu znakowego, natomiast metoda `lastIndexOf` szuka ostatniego takiego wystąpienia. Zobaczmy prosty przykład:

```
const str = 'abcabc';
str.indexOf('b');     // 1
str.lastIndexOf('b'); // 4
```

Litera `b` znajduje się w ciągu w dwóch miejscach, na pozycji o indeksie `1` oraz `4`. Widzimy tutaj wyraźnie jak działają metody `indexOf` oraz `lastIndexOf`. Podobnie jak metody omawiane wcześniej, ta również uwzględnia wielkość znaków, dlatego próba wyszukania wielkiej litery `B` kończy się niepowodzeniem:

```
str.lastIndexOf('B'); // -1
```

Powyższe użycie metody `lastIndexOf` jest dość zrozumiałe i intuicyjne. Problem pojawia się jednak, gdy zechcemy przekazać jej drugi parametr. O ile w metodzie `indexOf` jego działanie było zrozumiałe, oznaczało po prostu indeks, od którego mamy rozpocząć szukanie, o tyle w metodzie `lastIndexOf` jego działanie jest nieco bardziej skomplikowane.

Parametr ten oznacza bowiem indeks ostatniego znaku, który ma być uważany za początek analizowanego ciągu. Brzmi to zapewne nieco skomplikowanie, dlatego przeanalizujmy kilka przykładów, zapożyczonych wprost z dokumentacji MDN:

```
'canal'.lastIndexOf('a');    //3
'canal'.lastIndexOf('a', 2); //1
'canal'.lastIndexOf('a', 0); //-1
```

Gdy nie podaliśmy drugiego parametru, metoda analizowała cały ciąg znakowy. W drugim przypadku wskazaliśmy jednak, że ostatnim znakiem ma być znak o indeksie `2`, czyli litera `n`. Z tego powodu metoda `lastIndexOf` nie uwzględniła w analizie liter `al`, czyli dwóch ostatnich znaków słowa `canal`, i znalazła tylko jedną literę `a` na pozycji `1`. Gdy wskazaliśmy jako indeks wartość zerową, metoda nie dopasowała żadnego znaku, ponieważ ograniczyliśmy analizowany ciąg do 0 znaków.

Jeśli wyszukujemy pojedyncze znaki, to działanie drugiego parametru, po chwili zastanowienia się prawdopodobnie wydaje się całkiem sensowne i zrozumiałe. Czas więc zrobić mały mętlik w głowie i sprawdzić dwa kolejne przykłady:

```
'canal'.lastIndexOf('a', 3);  //3
'canal'.lastIndexOf('al', 3); //3
```

W pierwszym przypadku zakładamy, że interesują nas tylko pierwsze cztery znaki (indeksy od 0 do 3), czyli litery `cana`. Zgodnie z oczekiwaniami ostatnie wystąpienie litery `a` jest w tym wypadku dokładnie na końcu analizowanego ciągu, czyli na indeksie `3`.

Ale dlaczego w takim razie drugi przykład również odnalazł ciąg `al` na pozycji `3`? Przecież dopiero co mówiliśmy, że indeks wskazuje na ostatni znak, jaki mamy analizować…

Otóż tutaj musimy nieco sprostować wcześniejszą definicję. Indeks przekazany jako drugi parametr do metody `lastIndexOf` oznacza indeks ostatniego znaku w analizowanym ciągu, jednakże odnosi się wyłącznie do pozycji pierwszego znaku z szukanego ciągu. Odnosząc to do powyższego przykładu, indeks `3` oznacza, że pierwszy znak z szukanego fragmentu `al` (czyli pozycja litery `a`) może znaleźć się nie dalej, niż na indeksie trzecim w całym ciągu `canal`.

Jak widzisz stosowanie drugiego parametru w metodach `indexOf` oraz `lastIndexOf` może czasami być nie do końca intuicyjne, dlatego raczej zalecam unikać tego typu wywołań tych metod. Jeśli okaże się, że z jakichś powodów faktycznie musisz skrócić analizowany ciąg, to najbezpieczniej po prostu wywołać na nim wcześniej metodę `slice` i na otrzymanym ciągu użyć jednej z omawianych w tym rozdziale metod. Jeśli zdecydujesz się na używanie indeksów w tych metodach to pamiętaj, aby dokładnie przeanalizować, czy naprawdę poprawnie je zastosowałeś i czy wyniki są zgodne z oczekiwaniami — szczególnie gdy używasz drugiego parametru w metodzie `lastIndexOf`.

W zastosowaniach praktycznych w większości sytuacji zapewne w zupełności wystarczająca będzie metoda `includes`, zwracająca wartość `boolean`. Używając metod `indexOf` lub `lastIndexOf` pamiętaj, że w przypadku wykorzystania ich jako zamiennika metody `includes` należy wynik zawsze porównywać z wartością `-1`.

Podział ciągu na tablicę

W kolejnym rozdziale przyjrzymy się bliżej tablicom w języku JavaScript, z których korzysta się praktycznie w każdej aplikacji. Zanim jednak przejdziemy do tablic, cofnijmy się nieco i przypomnijmy sobie, jak w JavaScript można odwoływać się do poszczególnych znaków w ciągach typu `string`:

```
const str = 'abcd';
str[0];     // "a"
str[3];     // "d"
str.length; // 4
```

Zapisy te przypominają nieco odwołania do elementów tablicy. Zobaczmy to na innym przykładzie, w którym w zmiennej `str` będziemy przechowywać tablicę, której elementami będą poszczególne litery `a`, `b`, `c`, `d`.

```
const str = ['a', 'b', 'c', 'd'];
str[0];     // "a"
str[3];     // "d"
str.length; // 4
```

Należy jednak uważać, ponieważ ciągi znakowe nie są tablicami, a tym samym nie dziedziczą prototypowo po `Array.prototype`, w którym to znajduje się wiele przydatnych metod (omówię je w następnym rozdziale). Analiza tych przykładów sugeruje jednak, że prawdopodobnie nie powinno stanowić problemu przekonwertowanie ciągu znakowego na tablicę i odwrotnie. I tak też jest; służy do tego metoda `String.prototype.split`. Przyjmuje ona jeden parametr, którym jest tzw. separator, czyli znak traktowany jako element rozdzielający poszczególne znaki na elementy tablicy:

```
const str = 'a-b-c-d';
str.split('-'); // ["a", "b", "c", "d"]
```

W roli separatora wskazaliśmy tutaj jawnie znak myślnika. Zwróć uwagę, że separator ten nie znajduje się w otrzymanej tablicy, jest on w pewnym sensie likwidowany. Jednocześnie jednak nasz ciąg oryginalny zapisany w stałej `str` nie ulega zmianie.

Separatorem nie musi być pojedynczy znak, równie dobrze może to być kilka znaków:

```
const str = 'a---b---c---d';
str.split('---'); // ["a", "b", "c", "d"]
```

W przypadku, gdy nie uda się znaleźć ani jednego wystąpienia wskazanego separatora, metoda zwraca tablicę zawierającą jeden element, którym jest cały ciąg znakowy. Jest to dobre rozwiązanie, ponieważ wywołując metodę `split` na jakimkolwiek ciągu znakowym, możemy mieć pewność, że otrzymana wartość zawsze będzie tablicą:

```
str.split('x'); // ["a---b---c---d"]
```

Ciekawym przypadkiem użycia metody `split`, który z pewnością spotkasz czasami w starszych aplikacjach, jest podanie w roli separatora pustego ciągu znakowego, co skutkuje wyodrębnieniem każdego znaku jako osobnego elementu tablicy:

```
const str = 'abcd'
str.split(''); // ["a", "b", "c", "d"]
```

W nowych wersjach języka JavaScript zapis ten można uprościć za pomocą operatora `spread`, który również zwróci tablicę pojedynczych elementów:

```
const strAsArray = [ ...str ];
strAsArray; // ["a", "b", "c", "d"]
```

Metoda `split` może przyjmować również drugi, opcjonalny parametr, którym jest limit, określający maksymalną liczbę elementów w otrzymanej tablicy. Załóżmy, że chcemy dokonać podziału ciągu `abcd` na tablicę zawierającą pojedyncze litery, jednakże interesują nas tylko pierwsze trzy elementy, czyli litery `abc`:

```
str.split('', 3); // ["a", "b", "c"]
```

W roli separatora metoda może przyjmować albo ciąg znakowy, albo tzw. wyrażenie regularne *regexp*. Wykorzystanie *regexp* daje nam nieco większą elastyczność w podziale, np. pozwala stosować alternatywnie kilka różnych separatorów. W niektórych sytuacjach wykorzystanie *regexp* w metodzie `split` może jednak kończyć się nie do końca zgodnie z oczekiwaniami, szczególnie przy bardziej złożonych wyrażeniach. W wielu aplikacjach można jednak spotkać się z typowym przypadkiem użycia *regexp* w metodzie `split`, gdy potrzebujemy rozdzielić wartość reprezentującą kwotę pieniężną na złotówki i grosze:

```
const price1 = '20,45';
const price2 = '20.45';
price1.split(/[,.]/); // ["20", "45"]
price2.split(/[,.]/); // ["20", "45"]
```

Powyższe wyrażenie regularne umożliwia podział albo po znaku kropki, albo po znaku przecinka. Jest to dobra praktyka, ponieważ wielu ludzi jest przyzwyczajonych do podawania kwot z użyciem przecinka, dlatego wykorzystanie takiego *regexp* gwarantuje nam, że uzyskamy złotówki i grosze niezależnie od tego, w jaki sposób użytkownik wprowadzi te dane w aplikacji.

JavaScript od podstaw!

- **Poznaj funkcje języka JavaScript**
- **Dowiedz się, co możesz z nim osiągnąć**
- **Naucz się wykorzystywać go w praktyce**

JavaScript to bardzo popularny język programowania, który z rozwiązania stosowanego niegdyś głównie do wykonywania prostych akcji na stronach internetowych ewoluował do postaci pełnoprawnego narzędzia. Z powodzeniem można wykorzystywać je w rozmaitych sytuacjach i na wielu różnych platformach do tworzenia całkiem zaawansowanych aplikacji. Uznanie użytkowników zapewniły JavaScriptowi jego prostota, doskonała dokumentacja oraz duże i stale powiększające się możliwości.

Jeśli zależy Ci na szybkim rozpoczęciu programowania w JavaScripcie, trafiłeś w dziesiątkę! Ta książka łatwo wprowadzi Cię w świat tego języka — zaprezentuje jego podstawowe konstrukcje i funkcje, przedstawi możliwości programowania obiektowego oraz pokaże, jak posługiwać się liczbami, tablicami, datami i ciągami znakowymi. Zdobytą wiedzę utrwalisz dzięki praktycznym ćwiczeniom, które nie tylko pomogą Ci opanować sposoby użycia poszczególnych mechanizmów, lecz również pozwolą zetknąć się z rzeczywistymi zastosowaniami języka w codziennej pracy.

- **Składnia języka**
- **Zmienne i stałe**
- **Operatory i instrukcje sterujące**
- **Funkcje i obiekty**
- **Klasy i metody**
- **Ciągi znakowe**
- **Tablice i operacje na nich**
- **Liczby i daty**
- **Ćwiczenia praktyczne**

Odkryj, naucz się, stosuj! Praktycznie z JavaScriptem!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-5637-5

Cena: 49,00 zł