JavaScript od pierwszej linii kodu

Błyskawiczna nauka pisania gier, stron WWW i aplikacji internetowych

Laurence Lars Svekis
Maaike van Putten
Rob Percival

Helion

Packt>

Tytuł oryginału: JavaScript from Beginner to Professional: Learn JavaScript quickly by building fun, interactive, and dynamic web apps, games, and pages

Tłumaczenie: Robert Górczyński

ISBN: 978-83-8322-197-7

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

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

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)

Printed in Poland.

Spis treści

12

ROZDZIAŁ

Średniozaawansowany JavaScript

Koncepcje i rozwiązania zaprezentowane dotychczas w książce nie są jedynymi sposobami radzenia sobie z problemami programistycznymi. W tym rozdziale rzucimy Ci wyzwanie i zmusimy do nieco większego wysiłku, większej ciekawości oraz praktykowania dobrych nawyków w zakresie optymalizacji rozwiązań.

W poprzednich rozdziałach obiecywaliśmy, że w późniejszych pojawi się omówienie doskonałych funkcjonalności. Optymalne użycie części wbudowanych metod wymaga znajomości wyrażeń regularnych, które zostaną omówione w tym rozdziale. Przed nami jeszcze wiele innych ciekawych tematów — poruszymy wymienione tutaj zagadnienia:

- Wyrażenia regularne.
- Funkcja i obiekt `arguments`.
- Hoisting w JavaScripcie.
- Tryb ścisły.
- Debugowanie.
- Używanie ciasteczek.
- Lokalny magazyn danych.
- JSON.

Jak widać, mamy tutaj wiele różnych tematów, a każdy z nich jest zaawansowany i interesujący. Poszczególne podrozdziały nie są ze sobą powiązane, jak miało to miejsce w poprzednich rozdziałach. To przede wszystkim odmienne zagadnienia, których opanowanie naprawdę pomoże Ci poszerzyć własne umiejętności i znacznie poprawić znajomość JavaScriptu.

Odpowiedzi do ćwiczeń i na pytania quizu oraz gotowe projekty znajdziesz w „Dodatku”.

Wyrażenia regularne

Wyrażenia regularne, określane również mianem **regex**, to po prostu sposoby opisywania wzorców tekstu. Możesz je uznawać za ciągi tekstowe następnej generacji. Istnieją różne implementacje wyrażeń regularnych. To oznacza, że w zależności od interpretera wyrażenie regularne może się nieco różnić w zapisie. Mimo tego wyrażenia regularne są w pewien sposób ustandaryzowane, zapisuje się je (praktycznie) więc w taki sam sposób niezależnie od implementacji. W tym podrozdziale pokażemy przykłady użycia wyrażeń regularnych w JavaScripcie.

Wyrażenia regularne okazują się niezwykle użyteczne w wielu sytuacjach, np. podczas wyszukiwania błędów w ogromnym pliku bądź w celu pobrania informacji o agencie używanej przeglądarki WWW. Zastosowanie znajdują również w trakcie weryfikacji formularza, ponieważ wyrażenie regularne pozwala na określenie prawidłowych wzorców dla pól danych, np. adresu e-mail lub numeru telefonu.

Wyrażenie regularne okazuje się użyteczne nie tylko podczas wyszukiwania ciągów tekstowych, ale również w trakcie ich zastępowania. Być może myślisz teraz: *wyrażenia regularne są zadziwiające, ale gdzie tkwi haczyk?*. Masz rację, niestety jest i haczyk. Przede wszystkim zapisane wyrażenie regularne może wyglądać, jakby kot sąsiada przeszedł po klawiaturze i przypadkowo nacisnął losowo wybrane klawisze. Spójrz na przykład wyrażenia regularnego, które sprawdza poprawność adresu e-mail:

```
/([a-zA-ZO-9._-]+@[a-zA-ZO-9._-]+\.[a-zA-ZO-9._-]+)/g
```

Bez obaw! Po zakończeniu lektury tego podrozdziału będziesz w stanie odszyfrować tajne wzorce wyrażeń regularnych. Nie zamierzamy w tym miejscu omawiać wszystkich kwestii związanych z wyrażeniami regularnymi. Przedstawimy jednak solidne podstawy, które pozwolą Ci rozpocząć pracę z nimi i później pogłębić wiedzę w tym zakresie.

Rozpoczynamy od prostego przykładu. Wyrażenie regularne jest definiowane między dwoma ukośnikami. Oto przykład poprawnego wyrażenia regularnego:

```
/JavaScript/
```

To wyrażenie znajdzie dopasowanie, jeśli analizowany ciąg tekstowy zawiera słowo `JavaScript`. W przypadku dopasowania wynik jest pozytywny. Następnie ten wynik można wykorzystać do wielu celów.

Można do tego użyć wbudowanej funkcji JavaScriptu o nazwie `match()`. Zwraca ona dopasowanie wyrażenia regularnego (o ile zostało znalezione) oraz m.in. indeks początkowy tego dopasowania w analizowanym ciągu tekstowym.

Istnieje jeszcze inna funkcja wbudowana używająca wyrażenia regularnego, ale do niej przejdziemy nieco później — `match()` to po prostu wygodna funkcja podczas prezentowania wyrażeń regularnych. W kolejnym fragmencie kodu możesz zobaczyć ją w akcji.

```
let text = "Uwielbiam JavaScript!";
console.log(text.match(/javascript/));
```

Wynikiem jego wykonania jest `null`; ponieważ domyślnie wielkość liter ma znaczenie, stąd brak dopasowania słowa `javascript`. Jeżeli szukanym ciągiem tekstowym byłby `/ava/` lub po prostu `/a/`, dopasowanie zostałoby znalezione, ponieważ zmienna `text` zawiera `ava` i `a`. Jeżeli nie chcesz uwzględniać wielkości liter, możesz to wskazać za pomocą litery `i` po ukośniku kończącym wyrażenie regularne. W kolejnym poleceniu wyrażenie spowoduje dopasowanie słowa `javascript` w ciągu tekstowym `text`.

```
console.log(text.match(/javascript/i));
```

To polecenie wyświetli wynik w konsoli. Teraz wielkość liter nie ma znaczenia i dlatego słowo `javascript` zostało dopasowane. Spójrz na wynik wygenerowany przez to polecenie:

```
[
  'JavaScript',
  index: 10,
  input: 'Uwielbiam JavaScript!',
  groups: undefined
]
```

Wynikami są: obiekt zawierający znalezione dopasowanie, indeksy, w których rozpoczyna się to dopasowanie, a także sprawdzone dane wejściowe. W omawianym przykładzie grupy pozostają niezdefiniowane. Grupy można tworzyć za pomocą nawiasów okrągłych, co wyjaśnimy w dalszej części podrozdziału.

W JavaScripcie wyrażenia regularne są często stosowane w połączeniu z wbudowanymi metodami wyszukiwania i zastępowania ciągów tekstowych. Dokładniej omówimy to w następnym punkcie.

Określanie wielu opcji dla słów

Aby zdefiniować określony zakres opcji, można skorzystać z pokazanej tutaj składni:

```
let text = "Uwielbiam JavaScript!";
console.log(text.match(/javascript|nodejs|react/i));
```

W tym przypadku wyrażenie regularne dopasuje słowa `javascript`, `nodejs` lub `react`. W naszym źródłowym ciągu tekstowym dopasowane jest tylko pierwsze wystąpienie dowolnego z wymienionych słów, a następnie wyrażenie regularne kończy działanie. Dlatego też nie zostaną znalezione dwa dopasowania ani ich większa liczba — kolejny fragment kodu spowoduje wygenerowanie takich samych danych wyjściowych jak poprzednio.

```
let text = "Uwielbiam React i JavaScript!";
console.log(text.match(/javascript|nodejs|react/i));
```

Oto wygenerowane dane wyjściowe:

```
[
  'React',
  index: 10,
  input: 'Uwielbiam React i JavaScript!',
  groups: undefined
]
```

Jeżeli chcesz znaleźć wszystkie dopasowania, musisz użyć modyfikatora globalnego `g`. Sposób jego działania jest podobny do wyszukiwania bez uwzględniania wielkości liter. W omawianym przykładzie szukamy wszystkich dopasowań, niezależnie od wielkości liter. Wszystkie modyfikatory podaje się po ostatnim ukośniku. Jednocześnie można używać wielu modyfikatorów, jak pokazaliśmy w kolejnym fragmencie kodu. To równie dobrze może być tylko jeden, np. `g`.

```
let text = "Uwielbiam React i JavaScript!";
console.log(text.match(/javascript|nodejs|react/gi));
```

Ten kod zwróci w wyniku dopasowane słowa `React` i `JavaScript`.

```
[ 'React', 'JavaScript' ]
```

Jak widać, teraz wynik jest odmienny. Po użyciu specyfikatora `g` funkcja `match()` po prostu zwraca tablicę dopasowanych słów. Tutaj to nie jest ekscytujące, ponieważ tablica zawiera wskazane słowa. Znacznie większe zaskoczenie może być w przypadku bardziej złożonych wzorców. Dokładnie tym się zajmiemy w następnym punkcie.

Opcje znaków

Prezentowane dotychczas wyrażenia regularne były całkiem czytelne, prawda? Opcje znakowe powodują, że wyrażenia regularne zaczynają wyglądać, cóż, tajemniczo. Załóżmy, że chcesz znaleźć ciąg tekstowy składający się tylko z jednego znaku — `a`, `b` lub `c`. W takim przypadku możesz to zapisać następująco:

```
let text = "d";
console.log(text.match(/[abc]/));
```

Wartością zwrotną jest `null`, ponieważ ciąg tekstowy `d` nie zawiera znaku `a`, `b` lub `c`. Do wzorca możemy dodać literę `d`:

```
console.log(text.match(/[abcd]/));
```

Teraz zostaną wygenerowane już zupełnie inne dane wyjściowe.

```
[ 'd', index: 0, input: 'd', groups: undefined ]
```

Ponieważ szukamy zakresu znaków, możemy go zapisać w skróconej postaci:

```
let text = "d";
console.log(text.match(/[a-d]/));
```

Jeżeli chcesz znaleźć każdą literę, małą i wielką, możesz skorzystać z następującego wzorca:

```
let text = "t";
console.log(text.match(/[a-zA-Z]/));
```

Ten sam efekt można byłoby osiągnąć przez użycie modyfikatora wskazującego, że wielkość znaków nie ma znaczenia. Taki modyfikator ma zastosowanie dla wzorca jako całości, ale możesz go potrzebować jedynie w przypadku określonego znaku.

```
console.log(text.match(/[a-z]/i));
```

Dopasowanie zostanie znalezione dla obu powyższych wyrażeń regularnych. Jeżeli chcesz uwzględnić także cyfry, wówczas wyrażenie będzie miało następującą postać:

```
console.log(text.match(/[a-zA-Z0-9]/));
```

Jak widać, przeprowadzamy konkatenację zakresów w celu dopasowania jednego znaku. W podobny sposób można przeprowadzać konkatenację opcji dla konkretnego znaku, np. `[abc]`. W powyższym wyrażeniu regularnym zostały wymienione trzy możliwe zakresy. To wyrażenie regularne dopasuje małą lub wielką literę z zakresu od A do Z oraz dowolną cyfrę od 0 do 9.

To nie oznacza, że dopasuje tylko jednoznakowe ciągi tekstowe. Znajdzie również tylko pierwszy dopasowany znak w wielkoznakowych ciągach tekstowych, ponieważ nie został użyty modyfikator globalny. Natomiast znaki specjalne nie zostaną dopasowane.

```
let text = "äé!";
console.log(text.match(/[a-zA-Z0-9]/));
```

Aby pomóc w usunięciu trudności związanej z niedopasowywaniem skomplikowanych znaków do wyrażenia, kropka działa w charakterze specjalnego znaku wieloznacznego, który może dopasować dowolny znak. Jak sądzisz, jaki jest wynik działania kolejnego wyrażenia regularnego?

```
let text = "To dowolny tekst.";
console.log(text.match(/./g));
```

Skoro został użyty modyfikator globalny, wyrażenie dopasuje dowolny znak. Spójrz na wygenerowane dane wyjściowe:

```
[
  'T', 'o', ' ', 'd', 'o',
  'w', 'o', 'l', 'n', 'y',
  ' ', 't', 'e', 'k', 's',
  't', '.'
]
```

Co można zrobić w sytuacji, gdy chce się dopasować jedynie kropkę? Jeżeli chcesz, aby znak specjalny (czyli ten używany w wyrażeniu regularnym do zdefiniowania wzorca) miał swoje zwykłe znaczenie lub jeśli zwykły znak ma mieć specjalne znaczenie, to trzeba go poprzedzić ukośnikiem.

```
let text = "To dowolny tekst.";
console.log(text.match(/\./g));
```

W tym przykładzie kropka została poprzedzona ukośnikiem. Dlatego też nie działa już jako znak wieloznaczny, a jest interpretowana jako literał kropki. Oto wynik wykonania tego wyrażenia regularnego:

```
[ '.' ]
```

Są pewne zwykłe znaki, które nabierają znaczenia specjalnego po poprzedzeniu ich ukośnikiem. Nie zamierzamy tego dokładnie tutaj omawiać, a ograniczymy się do przedstawienia kilku przykładów.

```
let text = "Mam 29 lat.";
console.log(text.match(/\d/g));
```

Jeżeli litera d zostanie poprzedzona ukośnikiem, \d, wówczas dopasuje dowolną cyfrę. Tutaj przeprowadzamy dopasowanie globalne, więc zostają dopasowane wszystkie cyfry.

```
[ '2', '9' ]
```

Ukośnikiem można poprzedzić także literę s, \s, i wówczas zostaną dopasowane białe znaki.

```
let text = "Programowanie daje naprawdę dużo satysfakcji!";
console.log(text.match(/\s/g));
```

Wynikiem wykonania tego przykładu będzie kilka spacji. Uwzględniane są również tabulatory i inne rodzaje białych znaków.

```
[ ' ', ' ', ' ', ' ' ]
```

Bardzo użyteczną opcją jest \b, dopasowująca tekst tylko wtedy, gdy znajduje się na początku słowa. W kolejnym przykładzie wyrażenie nie dopasuje egzemplarza na w słowie następnym.

```
let text = "Na następnym skrzyżowaniu?";
console.log(text.match(/\bna/gi));
```

Oto dane wyjściowe wygenerowane przez ten przykład:

```
[ 'Na' ]
```

Co prawda można sprawdzić pod kątem znaków będących liczbami, ale metoda match() należy do obiektu typu String, więc implementujesz ją z użyciem zmiennych liczbowych, jak pokazaliśmy w kolejnym fragmencie kodu.

```
let nr = 357;
console.log(nr.match(/3/g));
```

To spowoduje wygenerowanie błędu TypeError informującego, że nr.match() nie jest funkcją.

Grupy

Istnieje wiele powodów do stosowania grup w wyrażeniach regularnych. Gdy chcesz dopasować grupę znaków, możesz ją ująć w nawias. Spójrz na kolejny przykład:

```
let text = "Uwielbiam JavaScript!";
console.log(text.match(/(uwielbiam|poznaję)\s(javascript|motyle)/gi));
```

W tym przypadku będą szukane słowa uwielbiam lub poznaję, następnie biały znak i dalej słowo javascript lub motyle. Szukane będą wszystkie wystąpienia, a wielkość liter nie ma znaczenia. To wyrażenie regularne zwróci następujące dane wyjściowe:

```
[ 'Uwielbiam JavaScript' ]
```

Z grubsza może ono dopasować cztery warianty. Niektóre z nich wydają się dotyczyć także nas:

- Uwielbiam motyle.
- Poznaję motyle.
- Uwielbiam JavaScript.
- Poznaję JavaScript.

Grupy okazują się bardzo potężne, o ile wie się, jak można je powtarzać. Pokażemy to w kolejnym przykładzie Bardzo często pojawia się konieczność powtórzenia określonego fragmentu wyrażenia regularnego. Wówczas mamy kilka rozwiązań. Na przykład jeśli mają zostać dopasowane cztery dowolne znaki alfanumeryczne w sekwencji, wyrażenie można zapisać w pokazanej tutaj postaci:

```
let text = "Uwielbiam JavaScript!";
console.log(text.match(/[a-zA-Z0-9][a-zA-Z0-9][a-zA-Z0-9][a-zA-Z0-9]/g));
```

To spowoduje wygenerowanie następujących danych wyjściowych:

```
[ "Uwie", "lbia", "Java", "Scri" ]
```

Mamy tutaj przykład okropnego sposobu na powtarzanie bloku. Zapoznaj się ze znacznie lepszymi rozwiązaniami. Jeżeli dana grupa ma wystąpić zero lub jeden raz, można użyć znaku zapytania. To sprawdza się w przypadku znaków opcjonalnych, jak pokazaliśmy w kolejnym fragmencie kodu.

```
let text = "Świetnie się spisujesz!";
console.log(text.match(/i?s/gi));
```

W tym wyrażeniu szukamy litery `s`, która może być poprzedzona literą `i`. Dlatego też zostaną wygenerowane przedstawione tutaj dane:

```
[ "s", "s", "is", "s" ]
```

Bezsprzecznie raz to nie jest przykład powtarzania się. Zobacz, jak to wygląda w przypadku większej liczby powtórzeń. Jeżeli chcesz znaleźć coś przynajmniej raz, może być znacznie więcej, użyj znaku plus. Spójrz na przykład:

```
let text = "123123123";
console.log(text.match(/(123)+/));
```

To wyrażenie regularne będzie dopasowywało grupę `123` jeden lub więcej razy. Ponieważ ten ciąg tekstowy znajduje się w analizowanym ciągu, dopasowanie zostanie znalezione. Oto wygenerowane dane wyjściowe:

```
[ '123123123', '123', index: 0, input: '123123123', groups: undefined ]
```

Dopasowany jest cały ciąg tekstowy, ponieważ w omawianym przykładzie to po prostu powtórzony ciąg tekstowy `123`. Zdarzają się również sytuacje, w których wyrażenie regularne ma być dopasowane dowolną liczbę razy, na co wskazuje gwiazdka. Spójrz na przykładowy wzorzec:

```
/(123)*a/
```

Tutaj ciąg tekstowy `123` zostanie dopasowany dowolną liczbę razy. Dopasowane więc będą np.:

- 123123123a,
- 123a,
- a,
- ba.

Trzeba również dodać, że można szukać dopasowania wzorca konkretną liczbę razy. W takich przypadkach stosuje się składnię `{min, max}`. Spójrz na kolejny przykład:

```
let text = "abcabcabc";
console.log(text.match(/(abc){1,2}/));
```

Wygenerowane będą następujące dane wyjściowe:

```
[ 'abcabc', 'abc', index: 0, input: 'abcabcabc', groups: undefined ]
```

Tak się dzieje, ponieważ ciąg tekstowy `abc` został dopasowany jedno- i dwukrotnie. Jak widać, mimo użycia grup w danych wyjściowych wciąż wartością `groups` jest `undefined`. Aby określić grupę, trzeba nadać jej nazwę. Pokazaliśmy to w kolejnym przykładzie.

```
let text = "Uwielbiam JavaScript!";
console.log(text.match(/(?<language>javascript)/i));
```

To spowoduje wyświetlenie następujących danych wyjściowych:

```
[
  'JavaScript',
  'JavaScript',
  index: 7,
  input: 'Uwielbiam JavaScript!',
  groups: [Object: null prototype] { language: 'JavaScript' }
]
```

Możliwości wyrażeń regularnych są znacznie większe. Te zaprezentowane tutaj powinny być wystarczające w wielu interesujących sytuacjach. W następnym punkcie przedstawimy kilka praktycznych zastosowań wyrażeń regularnych.

Praktyczne zastosowania wyrażeń regularnych

Wyrażenia regularne okazują się doskonałe w wielu sytuacjach — wszędzie tam, gdzie trzeba dopasować określone wzorce ciągów tekstowych. W tym punkcie wyjaśnimy, jak można używać wyrażeń regularnych w połączeniu z innymi metodami przeznaczonymi do obsługi ciągów tekstowych, a także jak je zastosować podczas weryfikacji adresów e-mail i adresów IPv4.

Wyszukiwanie i zastępowanie ciągów tekstowych

W rozdziale 8. poświęconym wbudowanym metodom JavaScriptu omówiliśmy metody przeznaczone do wyszukiwania i zastępowania ciągów tekstowych. Chcieliśmy wówczas, aby w trakcie operacji wyszukiwania wielkość liter nie miała znaczenia. Mamy świetną wiadomość — do tego celu można wykorzystać wyrażenia regularne.

```
let text = "To jest przykładowe rozwiązanie.";
console.log(text.search(/Rozwiązanie/i));
```

Dodanie modyfikatora `i` po ostatnim ukośniku powoduje zignorowanie różnicy między wielkimi i małymi literami. Ten fragment kodu zwróci wartość `20`, która wskazuje na indeks początkowy znalezionego dopasowania. Tego nie można byłoby zrobić za pomocą zwykłego ciągu tekstowego.

Jak sądzisz, czy można zmienić sposób działania metody zastępowania, używając wyrażenia regularnego tak, aby zastąpić wszystkie wystąpienia ciągu tekstowego zamiast tylko pierwszego? Również w tym przypadku z pomocą przychodzi modyfikator. Do tego celu służy modyfikator globalny `g`. Aby poznać różnicę, spójrz na wyrażenie bez tego modyfikatora `g`:

```
let text = "Programowanie daje satysfakcję. Programowanie oferuje wiele
możliwości.";
console.log(text.replace("Programowanie", "JavaScript"));
```

Oto dane wyjściowe wygenerowane przez ten kod:

```
JavaScript daje satysfakcję. Programowanie oferuje wiele możliwości.
```

Bez użycia wyrażenia regularnego zmianie ulega tylko pierwsze wystąpienie ciągu tekstowego. W kolejnym fragmencie kodu mamy to samo wyrażenie regularne, ale z modyfikatorem globalnym `g`.

```
let text = "Programowanie daje satysfakcję. Programowanie oferuje wiele
możliwości.";
console.log(text.replace(/Programowanie/g, "JavaScript"));
```

Oto wynik działania tego wyrażenia regularnego:

```
JavaScript daje satysfakcję. JavaScript oferuje wiele możliwości.
```

Jak możesz zobaczyć, zmienione zostały wszystkie wystąpienia szukanego ciągu tekstowego.

Ćwiczenie praktyczne 12.1

Wyszukiwanie i zastępowanie ciągów tekstowych. To ćwiczenie obejmuje zastępowanie ciągów tekstowych w podanym ciągu tekstowym. Pierwsze pole tekstowe zawiera znak przeznaczony do zastąpienia, natomiast drugie zawiera znaki zastępujące znak z pierwszego pola. Operacja zostanie przeprowadzona po kliknięciu przycisku.

Zamieszczony tutaj kod HTML wykorzystaj jako szablon wyjściowy i dodaj niezbędny kod JavaScript.

```
<!doctype html>
<html>

<head>
  <title>Pełny kurs JavaScriptu</title>
</head>

<body>
  <div id="output">Pełny kurs JavaScriptu</div>
  Szukany znak:
  <input id="sText" type="text">
  <br> Znaki zastępujące:
  <input id="rText" type="text">
  <br>
  <button>Zastąp</button>
  <script>
```

```
  </script>
</body>

</html>
```

Wykonaj wymienione tutaj kroki:

1. Za pomocą JavaScriptu pobierz każdy z trzech elementów strony i obiekty tych elementów przypisz zmiennym, aby można było łatwo odwoływać się do nich w kodzie.
2. Do przycisku dodaj komponent nasłuchujący zdarzeń, aby kliknięcie przycisku spowodowało wywołanie funkcji.
3. Utwórz funkcję o nazwie `lookup()`, która będzie wyszukiwała i zastępowała tekst w elemencie danych wyjściowych. Treści tekstowej tego elementu przypisz zmienną o nazwie `s`, a następnie wartość zastępującą tę szukaną przypisz innej zmiennej, `rt`.
4. Utwórz nowe wyrażenie regularne z wartością pierwszego pola tekstowego, które pozwoli na zastąpienie tekstu. Używając tego wyrażenia regularnego, sprawdź dopasowanie za pomocą metody `match()`. Opakuj to warunkiem, który spowoduje wykonanie bloku kodu tylko po znalezieniu dopasowania.
5. Jeżeli dopasowanie zostanie znalezione, funkcja `replace()` powinna przeprowadzić operację zastąpienia.
6. Uaktualnij obszar danych wyjściowych, używając nowo utworzonych danych.

Weryfikacja adresu e-mail

Aby utworzyć wzorzec wyrażenia regularnego, najpierw trzeba spróbować opisać ten wzorzec słowami. Adres e-mail składa się z pięciu części, w postaci [*nazwa*]@[*domena*].[*rozszerzenie*].

Oto te pięć części:

1. *nazwa* — jeden lub więcej znaków alfanumerycznych, podkreślenie, minus lub kropka.
2. `@` — literał znaku `@`.
3. *domena* — jeden lub więcej znaków alfanumerycznych, podkreślenie, minus lub kropka.
4. `.` — literał znaku `.`.
5. *rozszerzenie* — jeden lub więcej znaków alfanumerycznych, podkreślenie, minus lub kropka.

A oto poszczególne części zapisane w postaci wyrażenia regularnego:

1. `[a-zA-Z0-9._-]+`.
2. `@`.
3. `[a-zA-Z0-9._-]+`.
4. `\.` (pamiętaj, że w wyrażeniu regularnym kropka ma znaczenie specjalne, więc trzeba ją poprzedzić ukośnikiem).
5. `[a-zA-Z0-9._-]+`.

Po zebraniu wszystkiego w całość otrzymujemy:

```
/([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9._-]+)/g
```

Spójrz na to wyrażenie regularne w akcji:

```
let emailPattern = /([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9._-]+)/g;
let validEmail = "maaike_1234@email.com";
let invalidEmail = "maaike@mail@.com";
console.log(validEmail.match(emailPattern));
console.log(invalidEmail.match(emailPattern));
```

Przetestowaliśmy ten wzorzec na prawidłowym i nieprawidłowym adresie e-mail. Oto wygenerowane dane wyjściowe:

```
[ 'maaike_1234@email.com' ]
null
```

Jak widać, w przypadku prawidłowego adresu e-mail zostaje on zwrócony, natomiast jeśli adres jest nieprawidłowy, wówczas wartością zwrotną jest `null` (brak dopasowania).

Ćwiczenie praktyczne 12.2

Utwórz aplikację używającą JavaScriptu do sprawdzenia, czy dane wejściowe w postaci ciągu tekstowego przedstawiają poprawnie sformatowany adres e-mail. Jako punkt wyjścia wykorzystaj zamieszczony tutaj szablon HTML.

```
<!doctype html>
<html>
<head>
  <title>Kurs JavaScriptu</title>
</head>
<body>
  <div class="output"></div>
  <input type="text" placeholder="Podaj adres e-mail">
  <button>Sprawdź</button>
  <script>

  </script>
</body>
</html>
```

Wykonaj wymienione tutaj kroki:

1. Zaprezentowanego szablonu HTML użyj jako punktu wyjścia podczas tworzenia aplikacji. W kodzie JavaScript pobierz elementy `<input>`, `<div>` i `<button>` strony jako obiekty JavaScriptu.
2. Do przycisku dodaj komponent nasłuchujący zdarzeń, aby kliknięcie przycisku spowodowało wykonanie bloku kodu, który pobierze wartość bieżącą w polu `<input>`. Utwórz pustą wartość odpowiedzi, która po ostatecznym przygotowaniu zostanie umieszczona w elemencie `<div>`.

3. Dodaj kod sprawdzający, czy wartość pola `<input>` jest w formacie poprawnego adresu e-mail. Jeżeli wynikiem sprawdzenia jest `false`, odpowiedź powinna zawierać komunikat `Nieprawidłowy adres e-mail`, a kolor tekstu tego komunikatu powinien być czerwony.
4. Jeżeli wynik sprawdzenia zwraca `true`, należy wyświetlić komunikat potwierdzający poprawność adresu e-mail, a kolor tekstu tego komunikatu ma być zielony.
5. Przygotowany komunikat wyświetl w elemencie `<div>`.

Funkcja i obiekt arguments

JavaScript radzi sobie z argumentami funkcji przez ich dodanie do obiektu o nazwie `arguments`. Ten obiekt działa podobnie jak tablica i można go użyć zamiast nazwy parametru. Spójrz na przedstawiony tutaj fragment kodu:

```
function test(a, b, c) {
  console.log("pierwszy:", a, arguments[0]);
  console.log("drugi:", b, arguments[1]);
  console.log("trzeci:", c, arguments[2]);
}

test("zabawa", "js", "tajemnice");
```

Ten kod wygeneruje następujące dane wyjściowe:

```
pierwszy: zabawa zabawa
drugi: js js
trzeci: tajemnice tajemnice
```

Po uaktualnieniu jednego z parametrów argumenty zostają odpowiednio zmienione. Tak samo dzieje się w przeciwnym kierunku.

```
function test(a, b, c) {
  a = "przyjemny";
  arguments[1] = "JavaScript";
  console.log("pierwszy:", a, arguments[0]);
  console.log("drugi:", b, arguments[1]);
  console.log("trzeci:", c, arguments[2]);
}

test("zabawa", "js", "tajemnice");
```

W tym kodzie zostaną zmienione `arguments[0]` i `b`, ponieważ są powiązane z, odpowiednio, `a` i `arguments[1]`, o czym możesz się przekonać, patrząc na wygenerowane dane wyjściowe.

```
pierwszy: przyjemny przyjemny
drugi: JavaScript JavaScript
trzeci: tajemnice tajemnice
```

Jeżeli funkcja zostanie wywołana z większą liczbą argumentów, niż została zadeklarowana w sygnaturze funkcji, można uzyskać do nich dostęp właśnie w taki sposób. Jednak nowoczesnym podejściem jest użycie parametru resztowego (`...parametr`) zamiast obiektu `arguments`.

Jeśli nie pamiętasz, czym jest parametr resztowy, możesz to sobie przypomnieć, wracając do rozdziału 6.

Ćwiczenie praktyczne 12.3

To ćwiczenie pokazuje użycie przypominającego tablicę obiektu `arguments` i wyodrębnianie z niego wartości. Użycie właściwości `length` tego obiektu umożliwia iterację przez argumenty i zwrot ostatniego elementu listy.

1. Utwórz funkcję bez żadnych parametrów. Zdefiniuj pętlę przeznaczoną do iteracji przez obiekt `arguments`. Dzięki temu będzie można przeprowadzić iterację przez wszystkie argumenty funkcji.
2. Utwórz zmienną o nazwie `lastOne` i nie przypisuj jej żadnej wartości.
3. Podczas iteracji przez argumenty zmiennej `lastOne` przypisz wartość bieżącą argumentu, używając indeksu `i` do zwrotu wartości argumentu. Argument będzie miał wartość indeksu, którą można wykorzystać w celu odniesienia się do tej wartości podczas iteracji przez obiekt `arguments`.
4. Zwróć wartość `lastOne`, która powinna zawierać tylko wartość ostatniego argumentu.
5. Wyświetl wartość odpowiedź wygenerowaną przez funkcję, przekaż funkcji pewną liczbę argumentów i wyświetl wynik w konsoli. Wyświetlony powinien być tylko ostatni element listy. Jeżeli chcesz zobaczyć wszystkie elementy, możesz je wyświetlić oddzielnie w konsoli podczas iteracji bądź przygotować tablicę, która może zostać zwrócona. Następnie podczas iteracji przez argumenty dodawaj je do tablicy.

Hoisting w JavaScripcie

W rozdziale 6. poświęconym funkcjom wyjaśniliśmy, że mamy trzy różne rodzaje zmiennych — `const`, `let` i `var` — i zalecamy używanie `let` zamiast `var` ze względu na różnice w zasięgu. Powodem tego jest tzw. **hoisting** w JavaScripcie. Hoisting polega na przeniesieniu deklaracji zmiennych na początek zakresu, w którym zostały zdefiniowane. To pozwala wykonywać akcje, które są niedostępne w wielu innych językach programowania, i jest ku temu dobry powód. Ten fragment kodu nie wzbudza żadnych podejrzeń.

```
var x;
x = 5;
console.log(x);
```

Jego działanie polega na wyświetleniu wartości `5`. Dzięki hoistingowi kolejny fragment kodu działa w taki sam sposób.

```
x = 5;
console.log(x);
var x;
```

Jeżeli spróbujesz to zrobić z użyciem słowa kluczowego `let`, otrzymasz błąd `ReferenceError`. Dlatego też lepiej jest stosować `let`. Kod taki jak w powyższym fragmencie jest trudny w odczycie i nieprzewidywalny, więc naprawdę lepiej go unikać.

Interpreter JavaScriptu przenosi wszystkie deklaracje `var` na początek pliku jeszcze przed rozpoczęciem jego przetwarzania. Przenoszone są tylko deklaracje, nie inicjacje. Dlatego jeśli spróbujesz użyć takiej zmiennej przed jej zainicjowaniem, otrzymasz wartość `undefined`. I dlatego też zmienna powinna być zainicjowana przed zadeklarowaniem. Taki projekt jest związany z alokacją pamięci, ale skutki uboczne pozostają niepożądane.

Istnieje jednak sposób na wyłączenie tego sposobu działania. W następnym podrozdziale dowiesz się, jak to zrobić.

Używanie trybu ścisłego

Luźny sposób działania JavaScriptu można do pewnego stopnia zmienić dzięki użyciu trybu ścisłego. Jego włączenie następuje po zastosowaniu w kodzie przedstawionego tutaj polecenia, które musi być pierwszym poleceniem w kodzie.

```
"use strict";
```

Spójrz na fragment kodu, który działa, gdy nie jest używany tryb ścisły:

```
function sayHi() {
  greeting = "Witaj!";
  console.log(greeting);
}

sayHi();
```

Zapomnieliśmy tutaj o zadeklarowaniu zmiennej `greeting`, więc JavaScript uzupełnia ten brak przez umieszczenie odpowiedniej deklaracji na początku pliku i wyświetla w konsoli komunikat `Witaj!`. Jednak włączenie trybu ścisłego powoduje wygenerowanie błędu:

```
"use strict";

function sayHi() {
  greeting = "Witaj!";
  console.log(greeting);
}

sayHi();
```

Ten fragment kodu prowadzi do następującego błędu:

```
ReferenceError: greeting is not defined
```

Trybu ścisłego można użyć tylko w określonej funkcji. Wystarczy wymienione wcześniej polecenie umieścić na początku funkcji, a tryb ścisły zostanie włączony tylko dla tej funkcji. Tryb ścisły oznacza również wprowadzenie kilku innych zmian. Na przykład mniejsza liczba słów może być zastosowana w nazwach zmiennych lub funkcji, ponieważ są one prawdopodobnie zarezerwowanymi słowami kluczowymi dla przyszłych wydań JavaScriptu.

Wykorzystanie trybu ścisłego to doskonały sposób używania JavaScriptu we frameworku, a nawet później, podczas tworzenia kodu TypeScriptu. Używanie trybu ścisłego uznaje się obecnie za dobrą praktykę i zachęcamy do tego podczas samodzielnego tworzenia kodu. Jednak w trakcie pracy z istniejącym, starszym kodem JavaScript często nie jest to łatwe rozwiązanie.

Po krótkim przedstawieniu trybu ścisłego nadeszła pora na zagłębienie się w zupełnie inny tryb: debugowania! Z trybem debugowania mamy do czynienia, gdy nie zajmujemy się tworzeniem lub uruchamianiem aplikacji, ale uruchomieniem jej w sposób specjalny, umożliwiający wyszukiwanie błędów.

Debugowanie

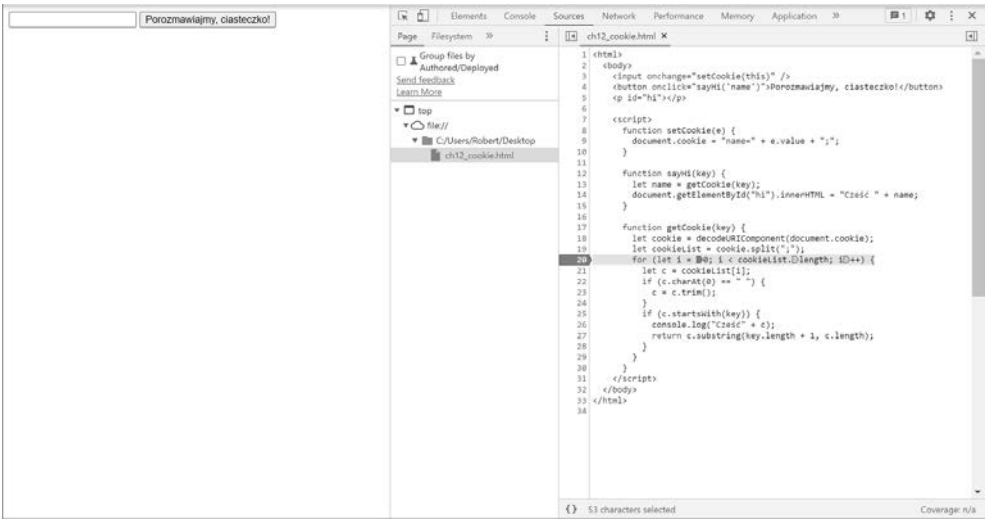
Debugowanie to delikatne zadanie. Na początku zwykle bardzo trudno jest ustalić, co złego dzieje się z kodem. Jeżeli używasz JavaScriptu w przeglądarce WWW i kod nie działa zgodnie z oczekiwaniami, wówczas pierwszym krokiem jest zawsze przejście do konsoli w przeglądarce WWW. Wyświetlone w niej komunikaty błędów bardzo często mogą pomóc ruszyć z miejsca.

Jeżeli to nie pomaga, w konsoli można wyświetlać informacje w trakcie każdego etapu wykonywania kodu, a także wartości zmiennych. W ten sposób zyskasz pewne informacje o tym, co się dzieje w kodzie. Być może działanie opiera się na określonej zmiennej, która okazuje się niezdefiniowana. Ewentualnie oczekujesz pewnej wartości z wyrażenia matematycznego, ale na skutek błędu wynik jest zupełnie inny, niż sądzisz. W trakcie pracy nad kodem JavaScript bardzo często używa się wywołań `console.log()`.

Punkt przerwania

Bardziej profesjonalne podejście do debugowania polega na użyciu punktów przerwania. Jest to możliwe w większości przeglądarek WWW i środowisk IDE. Wystarczy kliknąć wiersz przed podejrzanym poleceniem w kodzie (w przeglądarce WWW Chrome można to zrobić w panelu *Sources*, w innych przeglądarkach może to być inny panel), a pojawi się kropka lub strzałka. Po uruchomieniu aplikacji jej działanie zostanie wstrzymane w kliknięty wierszu, dając tym samym możliwość przeanalizowania wartości zmiennych i sprawdzenia kodu wiersz po wierszu.

W ten sposób masz możliwość zorientowania się, co się dzieje w kodzie i jak usunąć problem. W tym punkcie pokażemy, jak używać punktów kontrolnych w Chromie. Większość przeglądarek WWW działa podobnie. W Chromie przejdź do karty *Sources* panelu *Inspect*. Wybierz plik, w którym ma zostać zdefiniowany punkt przerwania. Następnie wystarczy kliknąć numer wiersza, a punkt przerwania zostanie ustawiony, jak pokazaliśmy na rysunku 12.1.



Rysunek 12.1. Punkt przerwania w przeglądarce WWW

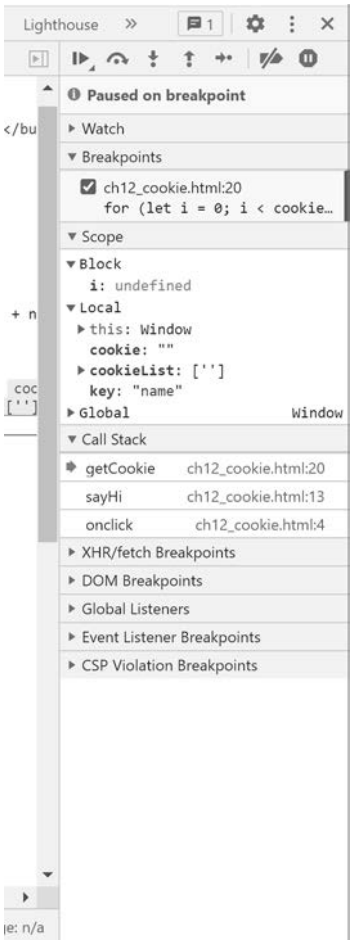
Teraz spróbuj wykonać wiersz kodu, w którym został zdefiniowany punkt przerwania. Zobaczysz, że wykonywanie kodu zostało wstrzymane. Po prawej stronie okna przeglądarki WWW można sprawdzić wartości wszystkich zmiennych, jak pokazaliśmy na rysunku 12.2.

Teraz możesz już przeanalizować kod wiersz po wierszu: kliknięcie na górze okna ikony strzałki skierowanej w prawo wznawia wykonywanie skryptu (będzie działał aż do napotkania następnego punktu przerwania bądź ponownie dotrze do tego samego punktu przerwania). Ikona przedstawiająca w połowie zakręconą strzałkę nad kropką powoduje przejście do następnego wiersza i pozwala przeanalizować istniejące wówczas wartości.

> Istnieje wiele różnych opcji punktów przerwania, których ze względu na ograniczoną ilość miejsca w rozdziale tutaj nie omówimy. Jeżeli chcesz dowiedzieć się więcej na temat debugowania kodu źródłowego z użyciem punktów przerwania, zapoznaj się z dokumentacją wybranego edytora kodu. Możesz również zajrzeć do dokumentacji Chrome'a pod adresem *https://developer.chrome.com/docs/devtools/javascript/breakpoints/*.

Ćwiczenie praktyczne 12.4

Podczas debugowania można śledzić wartości zmiennych w edytorze. W tym ćwiczeniu pokażemy, jak używać punktów przerwania edytora do sprawdzenia wartości zmiennej na pewnym etapie wykonywania skryptu. Wprawdzie tutaj mamy tylko prosty przykład, ale dokładnie ten sam proces może być używany podczas pobierania informacji z większych skryptów bądź w trakcie ustalania źródła problemu.



Rysunek 12.2. Analizowanie wartości zmiennych

> Występują pewne różnice i niuanse w sposobach, na jakie działają punkty przerwania w różnych edytorach. Dokładne informacje na ten temat znajdziesz w dokumentacji używanego środowiska. Tutaj chcemy jedynie ogólnie wyjaśnić ideę punktów przerwania i oferowanych przez nie możliwości w zakresie debugowania.

Jako przykład wykorzystamy następujący skrypt:

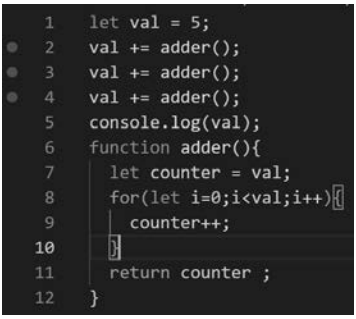
```
let val = 5;
val += adder();
val += adder();
val += adder();
console.log(val);
function adder(){
  let counter = val;
```

```
  for(let i=0;i<val;i++){
    counter++;
  }
  return counter ;
}
```

> Jeśli testujesz ten przykład w konsoli przeglądarki WWW, pamiętaj o dodaniu elementu `<script>` w kodzie i uruchomieniu pliku jako dokumentu HTML.

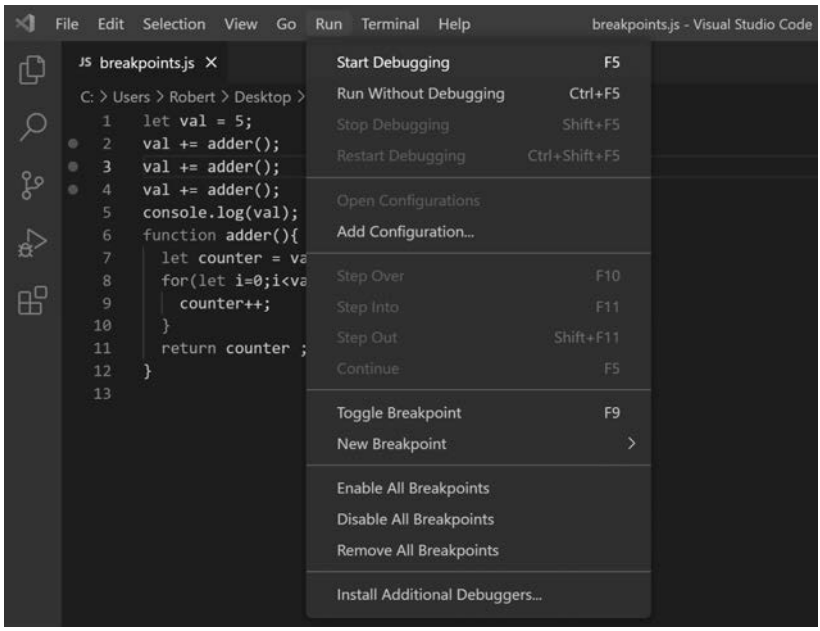
To ćwiczenie przetestowaliśmy w edytorze tekstu, ale równie dobrze można je wykonać w konsoli przeglądarki WWW bądź w innym środowisku.

1. Otwórz skrypt w wybranym edytorze bądź na karcie *Sources* w panelu *Inspect*. Kliknij po lewej stronie wiersza kodu, w którym chcesz umieścić punkt przerwania. Kropka wskazuje na zdefiniowanie punktu przerwania, jak pokazaliśmy na rysunku 12.3.

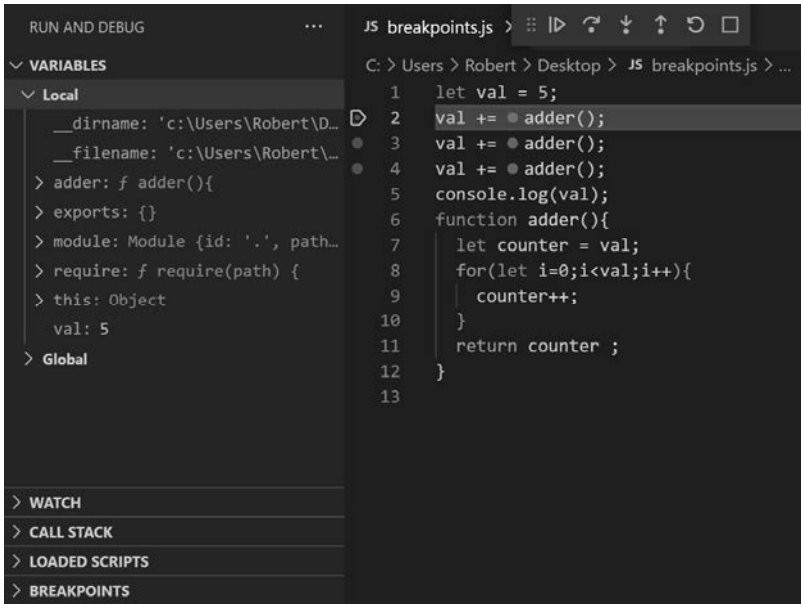


Rysunek 12.3. Zdefiniowane punkty przerwania

2. Uruchom kod, w którym znajdują się punkty przerwania. W omawianym przykładzie została wybrana opcja *Run/Start Debugging* (zobacz rysunek 12.4), ale to zależy od używanego edytora. Jeżeli korzystasz z konsoli przeglądarki WWW, możesz po prostu odświeżyć stronę internetową, aby w ten sposób ponownie uruchomić kod z nowo dodanym punktem przerwania.
3. Znajdziesz się w konsoli debugowania. Jedna z kart będzie zawierała listę wszystkich zmiennych w kodzie i ich wartości bieżące w punkcie przerwania. W moim edytorze ta karta nosi nazwę *VARIABLES*, natomiast w przeglądarce WWW Chrome to karta *Scope*.
4. Opcje menu pozwalają przejść do następnego punktu przerwania, zakończyć debugowanie bądź ponownie uruchomić sekwencję punktu przerwania. Kliknij ikonę przedstawiającą niebieski trójkąt skierowany w prawo, a przejdziesz do następnego punktu przerwania. Wartość zmiennej `val` wynosi `5`, zgodnie z poleceniem w wierszu 1. (zobacz rysunek 12.5). To jest pierwszy punkt przerwania w kodzie, po dotarciu do niego wykonywanie kodu zostanie wstrzymane. Pamiętaj, że zaznaczony wiersz nie został jeszcze wykonany.

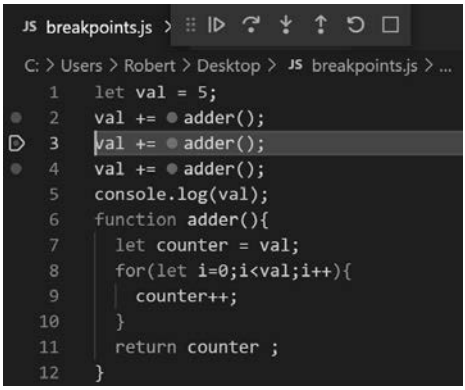


Rysunek 12.4. Ponowne uruchomienie kodu po dodaniu punktów przerwania



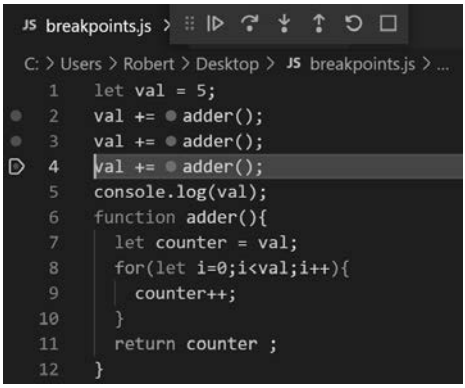
Rysunek 12.5. Wartości zmiennych w konsoli debugowania

5. Ponownie kliknij ten sam przycisk, a skrypt będzie wykonywany do następnego punktu przerwania (zobacz rysunek 12.6). W tym momencie wartość zmiennej zostanie uaktualniona, zgodnie z poleceniem w wierszu 2.



Rysunek 12.6. Poruszanie się między punktami przerwania w skrypcie

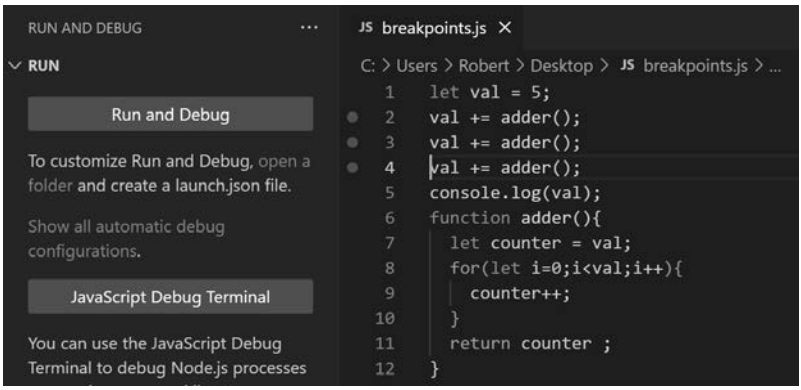
6. Kolejny raz kliknij ten sam przycisk, skrypt przejdzie do następnego punktu przerwania (zobacz rysunek 12.7), a wartość zmiennej znów zostanie uaktualniona.



Rysunek 12.7. Ostatni punkt przerwania w przykładowym kodzie

7. Po dotarciu do ostatniego punktu przerwania dostępne są jedynie opcje ponownego uruchomienia debugowania lub przejścia do debugowania w konsoli (zobacz rysunek 12.8). Jeżeli zdecydujesz się na tę pierwszą możliwość, zakończysz proces debugowania.

Ostateczna wartość zmiennej `val` po trzecim punkcie przerwania wyniosła 135. Zanotuj wartości tej zmiennej po pierwszym i drugim wywołaniu funkcji `adder()`, które można było poznać dzięki użyciu punktów przerwania.



Rysunek 12.8. Punkty przerwania w edytorze Visual Studio Code po wykonaniu ostatniego z nich

> To było proste ćwiczenie i zachęcamy do użycia punktów przerwania w trakcie analizy znacznie większych skryptów. Dzięki temu można lepiej zrozumieć, co się dzieje w kodzie po jego uruchomieniu.

Obsługa błędów

Dotychczas pokazaliśmy wiele błędów generowanych przez kod. Jak dotąd pozwalaliśmy, aby błąd powodował awarię programu. Istnieją jeszcze inne sposoby radzenia sobie z błędami. Gdy trzeba radzić sobie z kodem, którego działanie zależy od pewnych zewnętrznych danych wejściowych, np. pochodzących z API, od użytkownika lub z pliku, konieczne jest zapewnienie obsługi potencjalnych błędów powodowanych przez te dane wejściowe.

Jeżeli oczekujemy, że pewien fragment kodu zgłosi błąd, można umieścić go w bloku `catch`. W takim przypadku błąd zostanie przechwycony przez ten blok.

> Należy zachować ostrożność i nie nadużywać takiej możliwości. Nie musisz się tym martwić, gdy tworzysz lepszy kod i tym samym unikasz błędów w programie.

Oto przykład fragmentu kodu zgłaszającego błąd, umieszczonego w konstrukcji `try-catch`. Zakładamy, że funkcja `somethingVeryDangerous()` może zgłaszać błędy.

```
try {
  somethingVeryDangerous();
} catch (e) {
  if (e instanceof TypeError) {
    // Obsługa wyjątku typu TypeError.
  } else if (e instanceof RangeError) {
    // Obsługa wyjątku typu RangeError.
  } else if (e instanceof EvalError) {
```

```
    // Obsługa wyjątku typu EvalError.
  } else {
    // Obsługa wszystkich pozostałych typów wyjątku.
    throw e; // Ponowne zgłoszenie wyjątku.
  }
}
```

Jeżeli wystąpi błąd, zostanie przechwycony przez blok `catch`. Ponieważ `Error` może oznaczać wiele różnych błędów, sprawdzamy, który konkretnie błąd został zgłoszony, i tworzymy dla niego oddzielną procedurę obsługi. Do sprawdzenia klasy błędu używany jest operator `instanceof`. Po obsłużeniu błędu pozostała część kodu będzie wykonywana w zwykły sposób.

Możliwości konstrukcji `try-catch` są większe, np. można ją wzbogacić o blok `finally`. Kod umieszczony w tym bloku będzie wykonywany niezależnie od rodzaju zgłoszonego błędu. To świetne miejsce na kod przeprowadzający operacje porządkowe. Spójrz na przykład użycia konstrukcji `try-catch-finally`:

```
try {
  trySomething();
} catch (e) {
  console.log("O nie!");
} finally {
  console.log("Błąd czy nie błąd i tak będzie zarejestrowany!");
}
```

Dane wyjściowe tego kodu pozostają nieznane, ponieważ funkcja `trySomething()` nie została zdefiniowana. Jeżeli zgłosi błąd, w konsoli zostaną wyświetlone komunikaty, najpierw `O nie!`, a następnie `Błąd czy nie błąd i tak będzie zarejestrowany!`. Natomiast jeśli funkcja `trySomething()` nie wygeneruje błędu, wówczas zostanie wyświetlony tylko drugi z tych komunikatów.

Gdy z jakiegokolwiek powodu zachodzi potrzeba zgłoszenia błędu, można wykorzystać słowo kluczowe `throw`, jak pokazaliśmy w kolejnym fragmencie kodu.

```
function somethingVeryDangerous() {
  throw RangeError();
}
```

To może być doskonały sposób radzenia sobie z czymś, co pozostaje poza Twoją kontrolą, np. odpowiedzią pochodzącą z API, danymi wejściowymi użytkownika bądź danymi wejściowymi wczytanymi z pliku. Jeżeli zdarzy się coś nieoczekiwanego, poprawne obsłużenie takiej sytuacji czasami wymaga zgłoszenia błędu.

Ćwiczenie praktyczne 12.5

1. Używając słów kluczowych `throw`, `try` i `catch`, sprawdź, czy wartość jest liczbą. Jeżeli nie, zgłoś odpowiedni błąd.
2. Utwórz funkcję z argumentem o nazwie `val`.

3. W bloku `try` dodaj warunek sprawdzający, czy wartość `val` jest liczbą. Wykorzystaj do tego funkcję `isNaN()`. Jeżeli wartością zwrotną funkcji jest `true`, zgłoś błąd informujący, że dana wartość nie jest liczbą. W przeciwnym razie w konsoli powinien zostać wyświetlony komunikat `To jest liczba`.
4. Użyj bloku `catch` do przechwycenia błędów i wyświetlenia w konsoli informacji na ich temat.
5. Dodaj blok `finally` i gdy funkcja `isNaN()` zakończy działanie, wyświetl wartość zmiennej `val`.
6. Zdefiniuj dwa wywołania funkcji, pierwsze z argumentem w postaci ciągu tekstowego i drugie z argumentem liczbowym. Sprawdź dane wyjściowe wyświetlone w konsoli.

Praca z ciasteczkami

Ciasteczka to małe pliki danych przechowywane w Twoim komputerze i używane przez witryny internetowe. Zostały opracowane w celu przechowywania pewnych informacji o użytkowniku witryny internetowej. Ciasteczko ma postać ciągu tekstowego ze specjalnym wzorcem. Zawiera rozdzielone średnikami pary klucz-wartość.

Ciasteczko można utworzyć i korzystać z niego później. Spójrz na prosty przykład pokazujący utworzenie ciasteczka:

```
document.cookie = "name=Maaike;favoriteColor=black";
```

Po wydaniu po stronie klienta (np. w znaczniku `<script>`) to polecenie nie działa we wszystkich przeglądarkach WWW. Na przykład Chrome nie pozwala na tworzenie ciasteczka po stronie klienta. Kod musi być uruchomiony w serwerze. (W omawianym przykładzie użyłem przeglądarki WWW Safari, ale nie ma gwarancji, że ta funkcjonalność będzie obsługiwana w przyszłych wydaniach Safari). Alternatywą jest API internetowej pamięci masowej.

Istnieje możliwość uruchomienia Chrome'a z poziomu powłoki oraz z włączonymi pewnymi opcjami bądź też włączenie ciasteczek w ustawieniach dotyczących prywatności. Jeżeli nie będziesz dłużej potrzebować tej funkcjonalności, najlepiej jest ją wyłączyć. W kolejnym fragmencie kodu pokazaliśmy, jak można odczytać cookie.

```
let cookie = decodeURIComponent(document.cookie);
let cookieList = cookie.split(";");
for (let i = 0; i < cookieList.length; i++) {
  let c = cookieList[i];
  if (c.charAt(0) == " ") {
    c = c.trim();
  }
  if (c.startsWith("name")) {
    alert(c.substring(5, c.length));
  }
}
```

W tym przykładzie wszystkie ciasteczka zostały pobrane za pomocą metody `decodeURIComponent()` i następnie podzielone w miejscu średnika. W ten sposób powstała tablica, `cookieList`, z parami klucz-wartość w postaci ciągów tekstowych. Kolejnym krokiem jest iteracja przez wszystkie pary klucz-wartość. Później usuwamy białe znaki na początku i końcu wartości i sprawdzamy, czy rozpoczyna się od `name`. To jest nazwa ciasteczka użytego w tym kodzie.

Jeżeli chcesz pobrać wartość, musisz zacząć odczytywać po kluczu. W omawianym przykładzie długość klucza wynosi 4, czyli klucz kończy się w indeksie 3. Pomijamy znak równości w indeksie 4 i rozpoczynamy odczyt począwszy od indeksu 5. W tym kodzie imię zostało dodane do komunikatu wyświetlanego w oknie dialogowym. Spójrz na przykład prostej strony internetowej używającej ciasteczka do powitania użytkownika.

```
<!DOCTYPE html>
<html>
  <body>
    <input onchange="setCookie(this)" />
    <button onclick="sayHi('name')">Porozmawiajmy, ciasteczko!</button>
    <p id="hi"></p>

    <script>
      function setCookie(e) {
        document.cookie = "name=" + e.value + ";";
      }

      function sayHi(key) {
        let name = getCookie(key);
        document.getElementById("hi").innerHTML = "Cześć " + name;
      }

      function getCookie(key) {
        let cookie = decodeURIComponent(document.cookie);
        let cookieList = cookie.split(";");
        for (let i = 0; i < cookieList.length; i++) {
          let c = cookieList[i];
          if (c.charAt(0) == " ") {
            c = c.trim();
          }
          if (c.startsWith(key)) {
            console.log("Cześć" + c);
            return c.substring(key.length + 1, c.length);
          }
        }
      }
    </script>
  </body>
</html>
```

Jeżeli tworzysz nową witrynę internetową, unikaj takiego rozwiązania. Natomiast jeśli pracujesz ze starszym kodem, prawdopodobnie natkniesz się na takie podejście. Teraz już wiesz, na czym polega to rozwiązanie i jak je dostosować do własnych potrzeb. Bardzo dobrze!

Ćwiczenie praktyczne 12.6

Przystępujemy do opracowania projektu umożliwiającego utworzenie ciasteczka. Utwórz kilka funkcji przeznaczonych do pracy z ciasteczkami strony internetowej, m.in. odczytu wartości ciasteczka na podstawie jego nazwy, utworzenia nowego ciasteczka na podstawie nazwy i przypisania mu ważności przez pewną liczbę dni oraz usunięcia ciasteczka. Jako punkt wyjścia możesz wykorzystać przedstawiony tutaj szablon HTML.

```
<!doctype html>
<html>
<head>
  <title>Pełny kurs JavaScriptu</title>
</head>
<body>
  <script>

  </script>
</body>
</html>
```

Wykonaj omówione tutaj kroki:

1. Utwórz stronę internetową, a następnie w kodzie JavaScriptu wyświetl wartość `document.cookie`, w tym momencie to brak wartości.
2. Zdefiniuj funkcję pobierającą parametry dla `cookieName`, `cookieValue` i liczby dni ważności ciasteczka.
3. Sprawdź, czy wartość zmiennej `days` jest poprawna. W bloku poprawnego kodu pobierz bieżącą datę. Dla ciasteczka zdefiniuj datę jego ważności. Musi być wyrażona w milisekundach, więc skonwertuj wartość parametru dni na liczbę milisekund.
4. Zmodyfikuj obiekt daty, aby czas wygaśnięcia ważności ciasteczka był wartością w postaci ciągu tekstowego, która przedstawia czas UTC.
5. Skorzystaj z `document.cookie` w celu przypisania `cookieName = cookieValue`, dodaj informacje związane z datą ważności ciasteczka oraz określ `path=/`.
6. Zdefiniuj funkcję tworzącą ciasteczko testowe z pewną wartością i kilkudniową datą wygaśnięcia. W ten sam sposób utwórz drugie ciasteczko. Po odświeżeniu strony w konsoli powinny być wyświetlone informacje o przynajmniej dwóch ciasteczkach.
7. Zdefiniuj kolejną funkcję przeznaczoną do odczytania wartości ciasteczka. Zdefiniuj wartość `false`, a następnie utwórz tablicę ciasteczek rozdzielonych średnikami.
8. Przeprowadź iterację przez wszystkie ciasteczka i ponownie podziel je, tym razem w miejscu znaku równości. Tym samym pierwszy element, o indeksie `0`, będzie nazwą ciasteczka. Dodaj warunek sprawdzający, czy nazwa odpowiada żądanej w parametrach funkcji. W przypadku dopasowania przypisz wartość znajdującą się w drugim indeksie, czyli wartość ciasteczka o podanej nazwie. Funkcja powinna zwrócić `cookievalue`.

9. Za pomocą zdefiniowanej wcześniej funkcji odczytującej dwa ciasteczka wyświetl w konsoli dwa komunikaty, które powinny zawierać wartości ciasteczek.
10. W celu usunięcia ciasteczka musisz mu przypisać nieaktualną datę. Możesz utworzyć ciasteczko z datą `-1` i przekazać je razem z jego nazwą do funkcji tworzącej ciasteczko.
11. Spróbuj usunąć ciasteczko na podstawie jego nazwy.

Lokalny magazyn danych

Ciasteczka były postrzegane jako sposób na zachowanie danych użytkownika. Jednak istnieje znacznie nowocześniejsze rozwiązanie w tym zakresie: **lokalna pamięć masowa**. To jest zadziwiająca funkcjonalność, która pozwala tworzyć sprytnie działające witryny internetowe. Dzięki lokalnej pamięci masowej pary klucz-wartość można zapisywać w przeglądarce WWW i używać ich ponownie w nowej sesji (po późniejszym uruchomieniu przeglądarki WWW). Informacje są zwykle przechowywane w katalogu użytkownika, choć to zależy od przeglądarki WWW.

W ten sposób witryna internetowa zyskuje możliwość przechowywania pewnych informacji i ich późniejszego pobierania, nawet po odświeżeniu strony bądź zamknięciu przeglądarki WWW. W porównaniu z mechanizmem ciasteczek zaletą lokalnego magazynu danych jest to, że nie musi być przetwarzany w trakcie każdego żądania HTTP, jak ma to miejsce w przypadku ciasteczek. Lokalna pamięć masowa po prostu znajduje się na dysku i czeka, aż będzie potrzebna.

Obiekt `localStorage` to właściwość obiektu `window`, który już wcześniej był używany. Obiekt `localStorage` zawiera kilka metod, które trzeba znać, aby można było z niego efektywnie korzystać. Przede wszystkim potrzebne są metody pozwalające na pobranie i definiowanie par klucz-wartość z lokalnego magazynu danych. Aby coś zapisać w lokalnym magazynie danych, jest używana metoda `setItem()`, natomiast późniejsze pobranie tej wartości jest możliwe dzięki metodzie `getItem()`. Spójrz na przykład ich użycia:

```
<!DOCTYPE html>
<html>
  <body>
    <div id="stored"></div>
    <script>
      let message = "Powitaj lokalną pamięć masową!";
      localStorage.setItem("example", message);

      if (localStorage.getItem("example")) {
        document.getElementById("stored").innerHTML =
          localStorage.getItem("example");
      }
    </script>
  </body>
</html>
```

Ten fragment kodu powoduje wyświetlenie na stronie komunikatu `Powitaj lokalną pamięć masową!`. Dodawanie elementów do lokalnej pamięci masowej odbywa się przez podanie klucza i wartości w metodzie `setItem()`. Dostęp do tej pamięci można uzyskać bezpośrednio za pomocą `localStorage` bądź poprzez obiekt `window`. W omawianym przykładzie nazwa klucza to `example`, natomiast jego wartością zapisaną w pamięci masowej jest `Powitaj lokalną pamięć masową!`. Następnie kod sprawdza, czy klucz o podanej nazwie został utworzony w lokalnej pamięci masowej i wyświetla dane przez zapis wartości we właściwości `innerHTML` elementu `<div>` o identyfikatorze `stored`.

Jeżeli powrócisz do kodu i wyłączysz wiersz z wywołaniem `setItem()` przed wczytaniem strony po raz drugi, wartość nadal będzie wyświetlana, ponieważ w trakcie pierwszego wykonania programu informacje zostały zapisane w lokalnej pamięci masowej, a nie usunięte. Lokalna pamięć masowa nie ma daty wygaśnięcia, choć jej zawartość można usunąć ręcznie.

Klucz można również pobrać za pomocą indeksu. To użyteczne rozwiązanie, gdy trzeba przeprowadzić iterację przez pary klucz-wartość i nieznane są nazwy kluczy. W kolejnym poleceniu pokazaliśmy, jak można pobrać klucz, używając indeksu.

```
window.localStorage.key(0);
```

W tym przypadku kluczem jest `name`. W celu pobrania jego wartości można użyć następującego polecenia:

```
window.localStorage.getItem(window.localStorage.key(0));
```

Kolejne polecenie pokazuje, jak usunąć parę klucz-wartość:

```
window.localStorage.removeItem("name");
```

Wszystkie pary klucz-wartość można usunąć za pomocą pojedynczego wywołania:

```
window.localStorage.clear();
```

Dzięki lokalnemu magazynowi danych wartości mogą zostać zachowane nawet po zamknięciu przeglądarki WWW. To pozwala na implementacje wielu „sprytnych" rozwiązań, ponieważ aplikacja będzie w stanie „pamiętać" np. dane wprowadzone w formularzu, ustawienia wybrane w witrynie internetowej, a także to, które strony były wcześniej przeglądane.

Nie postrzegaj tych możliwości jako alternatywy dla obejście problemów związanych z ciasteczkami i prywatnością. Lokalna pamięć masowa rodzi dokładnie te same problemy co w przypadku ciasteczek, przy czym są one mniej znane. W witrynie internetowej nadal trzeba poinformować użytkownika o śledzeniu go i przechowywaniu informacji, podobnie jak ma to miejsce podczas obsługi ciasteczek.

Ćwiczenie praktyczne 12.7

Przystępujemy do utworzenia lokalnej pamięci masowej przeznaczonej do obsługi listy zakupów, której zawartość będzie przechowywana w przeglądarce WWW. Mamy tutaj przykład użycia JavaScriptu do konwersji z postaci ciągu tekstowego na użyteczny obiekt JavaScript oraz

z powrotem na ciąg tekstowy, który można przechowywać w lokalnej pamięci masowej. Jako punkt wyjścia możesz wykorzystać zamieszczony tutaj szablon.

```
<!doctype html>
<html>
<head>
  <title>JavaScript</title>
  <style>
    .ready {
      background-color: #ddd;
      color: red;
      text-decoration: line-through;
    }
  </style>
</head>
<body>
  <div class="main">
    <input placeholder="Nowy element" value="element testowy" maxlength="30">
    <button>Dodaj</button>
  </div>
  <ul class="output">
  </ul>
  <script>

  </script>
</body>
</html>
```

Wykonaj wymienione tutaj kroki:

1. W kodzie JavaScript wszystkie elementy strony pobierz jako obiekty JavaScriptu.
2. Utwórz tablicę `tasks` z wartością lokalnego magazynu danych `tasklist`, o ile istnieje. W przypadku braku magazynu danych tablica `tasks` powinna być pusta. Używając `JSON.parse()`, możesz skonwertować wartość ciągu tekstowego na postać obiektu w JavaScripcie.
3. Przeprowadź iterację przez wszystkie elementy tablicy `tasklist`. Będą przechowywane jako obiekty z nazwą i wartością boolowską określającą stan. Utwórz oddzielną funkcję przeznaczoną do utworzenia elementu zadania i dodania go z listy na stronę.
4. W funkcji generującej zadanie utwórz nowy element listy i `textNode`. Dołącz `textNode` do elementu listy, a następnie element umieść w obszarze danych wyjściowych na stronie. Jeżeli zadanie jest za pomocą wartości boolowskiej `true` określone jako wykonane, wówczas elementowi tego zadania dodaj klasę `ready`.
5. Do elementu zadania dodaj komponent nasłuchujący zdarzeń, który spowoduje przełączenie klasy `ready` po kliknięciu elementu. Za każdym razem gdy jest przeprowadzana zmiana dowolnego elementu listy, te zmiany trzeba przechowywać również w lokalnym magazynie danych. Utwórz funkcję, która będzie umieszczała te zmiany w lokalnym magazynie danych i gwarantowała, że dane wyświetlonej listy są takie same jak w magazynie danych. Konieczne jest usunięcie bieżącej zawartości tablicy listy zadań i odtworzenie jej na podstawie danych wizualnych. Utwórz więc odpowiedzialną za to funkcję.

6. Funkcja tworzenia zadań będzie usuwała bieżącą zawartość tablicy `tasks` i pobierała wszystkie elementy `<li>` na stronie. Trzeba przeprowadzić iterację przez każdy element listy, pobrać jego wartość tekstową i sprawdzić, czy zawiera klasę `ready`. Jeżeli tak, wówczas warunek stanu powinien mieć przypisaną wartość `true`. Wyniki dodaj do tablicy `tasks`. Dzięki temu zostanie ona ponownie utworzona od początku i będzie zawierała elementy widziane przez użytkownika na liście. Przekaż tablicę funkcji zapisującej zadania, aby zapisać ją w lokalnym magazynie danych. W ten sposób jeśli strona zostanie odświeżona, będzie zawierała tę samą listę.
7. W funkcji zapisującej zadania tablicę `tasks` przypisz `localstorage`. Konieczne jest konwertowanie obiektów na ciągi tekstowe, aby mogły być przypisywane przybierającemu postać ciągu tekstowego parametrowi lokalnego magazynu danych.
8. Teraz po odświeżeniu strony zobaczysz listę zadań. Kliknięcie zadania powoduje jego przekreślenie. Można również dodawać nowe zadania do listy — wystarczy wpisać zadanie w polu tekstowym i kliknąć przycisk *Dodaj*.

JSON

JSON (ang. *javascript object notation*) to format danych. Spotykaliśmy się już z nim podczas tworzenia obiektów w JavaScripcie. Mimo tego JSON nie oznacza obiektu JavaScriptu, to po prostu sposób na przedstawienie danych za pomocą formatu takiego jak w przypadku obiektu JavaScriptu. Dane w formacie można również łatwo skonwertować na obiekt JavaScriptu.

JSON to standard używany podczas komunikacji z API, w tym także API nieutworzonego w JavaScripcie. API może akceptować dane, np. pochodzące z formularza HTML w witrynie internetowej, w formacie JSON. Obecnie API niemalże zawsze udziela odpowiedzi w postaci danych w formacie JSON. Przekazywanie danych przez API następuje, np. gdy odwiedzasz sklep internetowy — informacje o oferowanych w nim produktach zwykle pochodzą z wywołania do API, które jest powiązane z bazą danych. Te dane są następnie konwertowane na format JSON i odsyłane do witryny internetowej. Spójrz na przykład danych w formacie JSON:

```
{
  "name" : "Malika",
  "age" : 50,
  "profession" : "programmer",
  "languages" : ["JavaScript", "C#", "Python"],
  "address" : {
    "street" : "Some street",
    "number" : 123,
    "zipcode" : "3850AA",
    "city" : "Utrecht",
    "country" : "The Netherlands"
  }
}
```

To jest obiekt, który, jak się wydaje, opisuje pewną osobę. Dane zawierają pary klucz-wartość. Klucz zawsze musi być ujęty w cudzysłów, natomiast w przypadku wartości w cudzysłów trzeba ujmować jedynie ciąg tekstowy. W omawianym przykładzie klucz to `name`, zaś wartość to `Malika`.

Lista wartości (czyli tablica JavaScriptu) jest wskazywana za pomocą nawiasu `[]`. Obiekt JSON zawiera listę `languages` (z wartościami w nawiasie kwadratowym) i inny obiekt, `address` (na co wskazuje nawias klamrowy).

Istnieje tylko kilka możliwych opcji związanych z formatem JSON:

- pary klucz-wartość z wartościami następujących typów: ciąg tekstowy, liczba, wartość boolowska i `null`;
- pary klucz-wartość z listami, czyli z nawiasami kwadratowymi, w których znajdują się elementy listy;
- pary klucz-wartość z innymi obiektami, czyli z nawiasami klamrowymi, w których znajdują się inne elementy JSON.

Te trzy opcje mogą być łączone, więc istnieje możliwość, że obiekt będzie zawierał inne obiekty, a lista inne listy. Przykład takiego rozwiązania możesz zobaczyć w poprzednim fragmencie kodu — obiekt zawiera zagnieżdżony obiekt `address`.

Takie zagnieżdżenie może być na jeszcze większym poziomie. Lista może zawierać obiekty, które z kolei mogą zawierać listy obiektów, listy list itd. To może wydawać się nieco skomplikowane i właśnie o to chodzi. Mimo że format jest bardzo prosty, to zagnieżdżanie wszystkich tych opcji może nieco skomplikować dane JSON. Nie bez powodu omówienie tego tematu znalazło się w rozdziale poświęconym zagadnieniom zaawansowanym.

Spójrz teraz na nieco bardziej skomplikowany przykład danych JSON:

```
{
  "companies": [
    {
      "name": "JavaScript Code Dojo",
      "addresses": [
        {
          "street": "123 Main street",
          "zipcode": 12345,
          "city" : "Scott"
        },
        {
          "street": "123 Side street",
          "zipcode": 35401,
          "city" : "Tuscaloosa"
        }
      ]
    },
    {
      "name": "Python Code Dojo",
      "addresses": [
        {
          "street": "123 Party street",
```

```
            "zipcode": 68863,
            "city" : "Nebraska"
          },
          {
            "street": "123 Monty street",
            "zipcode": 33306,
            "city" : "Florida"
          }
        ]
      }
    ]
}
```

Mamy tutaj listę firm, na której znalazły się dwa obiekty przedstawiające firmy. Te firmy mają dwie pary klucz-wartość: nazwę i listę adresów. Każda lista adresów przechowuje informacje o dwóch adresach, które z kolei składają się z trzech par klucz-wartość: `street`, `zipcode` i `city`.

Ćwiczenie praktyczne 12.8

To ćwiczenie pokaże, jak można utworzyć poprawny obiekt JSON, który będzie mógł być używany jako obiekt JavaScriptu. Przygotujesz prostą listę nazw i informacji o stanie, przez którą można przeprowadzić iterację i wyświetlić jej wynik w konsoli. Dane JSON wczytasz w kodzie JavaScript i wyświetlisz zawartość obiektu.

1. Utwórz obiekt JavaScriptu zawierający sformatowane dane JSON. Ten obiekt powinien zawierać przynajmniej dwa elementy, z których każdy powinien być obiektem z co najmniej dwoma parami klucz-wartość.
2. Utwórz funkcję wywoływaną w celu iteracji przez poszczególne elementy obiektu JavaScriptu JSON i wyświetlającą wynik w konsoli. Każdy element danych powinien zostać wyświetlony w konsoli za pomocą wywołania `console.log()`.
3. Wywołaj funkcję i uruchom kod JavaScript.

Przetwarzanie danych JSON

Istnieje wiele bibliotek i narzędzi przeznaczonych do przetwarzania ciągu tekstowego JSON na postać obiektu. Ciąg tekstowy JavaScriptu może zostać skonwertowany na obiekt JSON za pomocą funkcji `JSON.parse()`. Dane pochodzące z innej lokalizacji są zawsze uznawane za typ `String`, więc jeśli chcesz traktować je jako obiekt, musisz je wcześniej skonwertować. W kolejnym fragmencie kodu pokazaliśmy przykład takiej operacji.

```
let str = "{\"name\": \"Maaike\", \"age\": 30}";
let obj = JSON.parse(str);
console.log(obj.name, "ma", obj.age, "lat");
```

Dane po przetworzeniu mogą być traktowane jak obiekt. Dlatego też ten kod wyświetli w konsoli komunikat `Maaike ma 30 lat`.

Czasami konieczne jest przeprowadzenie odwrotnej operacji. Obiekt może być skonwertowany na ciąg tekstowy JSON za pomocą metody `JSON.stringify()`. Jej działanie polega na konwersji obiektu bądź wartości JavaScriptu na ciąg tekstowy JSON. W kolejnym fragmencie kodu możesz zobaczyć tę metodę w akcji.

```
let dog = {
  "name": "wiesje",
  "breed": "dachshund"
};

let strdog = JSON.stringify(dog);
console.log(typeof strdog);
console.log(strdog);
```

W wyniku przeprowadzonej konwersji typem `strdog` staje się ciąg tekstowy. Ta zmienna nie ma już dłużej właściwości `name` i `breed`, będą one niezdefiniowane. Przedstawiony kod spowoduje wyświetlenie w konsoli następujących danych wyjściowych:

```
string
{"name":"wiesje","breed":"dachshund"}
```

Takie rozwiązanie może być użyteczne np. w celu bezpośredniego przechowywania danych JSON w bazie danych.

Ćwiczenie praktyczne 12.9

To ćwiczenie pokazuje użycie metod JSON do przetwarzania danych JSON i konwertowania ciągów tekstowych na JSON. Dzięki wykorzystaniu metod JSON w połączeniu z JavaScriptem można skonwertować sformatowany ciąg tekstowy JSON na obiekt JavaScriptu oraz na odwrót.

1. Utwórz obiekt JSON z kilkoma elementami i obiektami. Możesz wykorzystać obiekt JSON z poprzedniego ćwiczenia.
2. Za pomocą metody JSON `stringify()` skonwertuj obiekt JSON JavaScript na ciąg tekstowy i przypisz go zmiennej `newStr [{"name":"Poznaj JavaScript","status":true},{"name":"Wypróbuj JSON","status":false}]`.
3. Używając metody `JSON.parse()`, skonwertuj wartość `newStr` z powrotem na obiekt i przypisz go zmiennej o nazwie `newObj`.
4. Przeprowadź iterację przez elementy `newObj` i wyświetl w konsoli wynik tej operacji.

Odpowiedź do ćwiczenia praktycznego 12.9

```
let myList = [{
  "name": "Poznaj JavaScript",
  "status": true
},
{
  "name": "Wypróbuj JSON",
  "status": false
}
```

```
];
const newStr = JSON.stringify(myList);
const newObj = JSON.parse(newStr);
newObj.forEach((el)=>{
  console.log(el);
});
```

Projekty rozdziału

Wyodrębnianie adresów e-mail

Zamieszczony tutaj kod HTML wykorzystaj jako szablon wyjściowy i dodaj do niego kod JavaScript, aby utworzyć funkcjonalność wyodrębniania adresów e-mail.

```
<!doctype html>
<html>
<head>
  <title>Pełny kurs JavaScriptu</title>
</head>
<body>
  <textarea name="txtarea" rows=2 cols=50></textarea> <button>Pobierz adresy e-
mail</button>
  <textarea name="txtarea2" rows=2 cols=50></textarea>
  <script>

  </script>
</body>
</html>
```

Wykonaj wymienione tutaj kroki:

1. W JavaScripcie pobierz pola tekstowe i przycisk jako obiekty JavaScriptu.
2. Do przycisku dodaj komponent nasłuchujący zdarzeń wywołujący funkcję, która pobierze zawartość pierwszego elementu `<textarea>` i zwróci adresy e-mail znalezione w tym elemencie.
3. W funkcji wyodrębniającej adresy e-mail pobierz zawartość pierwszego elementu `<textarea>`. Korzystając z wywołania `match()`, zwróć tablicę adresów e-mail dopasowanych w treści tego elementu.
4. W celu usunięcia duplikatów utwórz drugą tablicę, w której będą przechowywane jedynie unikatowe wartości.
5. Przeprowadź iterację przez wszystkie znalezione adresy e-mail i sprawdź, czy każdy z nich znajduje się już w tablicy `holder`. Jeżeli nie, dodaj go.
6. Używając metody tablicy `join()`, możesz połączyć ze sobą znalezione adresy e-mail i wyświetlić je w drugim elemencie `<textarea>`.

Weryfikacja formularza HTML

Ten projekt pokazuje przykład typowej struktury formularza, w którym sprawdza się wartości wprowadzone w polach i weryfikuje je przed wysłaniem formularza. Jeżeli wartości nie spełniają kryteriów zdefiniowanych w kodzie, użytkownikowi jest wyświetlany odpowiedni komunikat. Jako szablon wyjściowy wykorzystaj przedstawiony tutaj kod HTML i CSS.

```
<!doctype html>
<html>
<head>
  <title>Kurs JavaScriptu</title>
  <style>
    .hide {
      display: none;
    }
    .error {
      color: red;
      font-size: 0.8em;
      font-family: sans-serif;
      font-style: italic;
    }
    input {
      border-color: #ddd;
      width: 400px;
      display: block;
      font-size: 1.5em;
    }
  </style>
</head>
<body>
  <form name="myform"> E-mail :
    <input type="text" name="email"> <span class="error hide"></span>
    <br> Hasło :
    <input type="password" name="password"> <span class="error hide"></span>
    <br> Nazwa użytkownika :
    <input type="text" name="userName"> <span class="error hide"></span>
    <br>
    <input type="submit" value="Rejestruj"> </form>
  <script>

  </script>
</body>
</html>
```

Wykonaj wymienione tutaj kroki.

1. Używając JavaScriptu, pobierz wszystkie elementy strony i przypisz je zmiennym jako obiekty JavaScriptu, aby łatwiej pracować z nimi w kodzie. Ponadto pobierz wszystkie elementy strony o klasie `error` i zapisz je jako obiekt.
2. Do przycisku dodaj komponent nasłuchujący zdarzeń, aby przechwycić zdarzenie `click` i uniemożliwić domyślną akcję formularza.

3. Przeprowadź iterację przez wszystkie elementy strony o klasie `error` i dodaj im klasę `hide`, która spowoduje ich usunięcie z widoku, ponieważ mamy do czynienia z jeszcze niewypełnionym formularzem.
4. Używając wyrażenia regularnego dopasowującego poprawny adres e-mail, sprawdź wartość wprowadzoną w polu adresu e-mail.
5. Utwórz funkcję reagującą na błędy, która usunie klasę `hide` z elementu znajdującego się obok elementu wywołującego zdarzenie. W funkcji aktywuj także ten element.
6. Jeżeli istnieje błąd wskazujący na niedopasowanie danych wejściowych do określonego wyrażenia regularnego, przekaż parametry do utworzonej funkcji obsługi błędów.
7. Sprawdź pole hasła i upewnij się, że wprowadzona wartość zawiera jedynie litery i liczby. Sprawdź również, czy długość hasła mieści się w przedziale od 3 do 8 znaków. Jeżeli oba te warunki nie są spełnione, przygotuj komunikat błędu za pomocą utworzonej wcześniej funkcji obsługi błędów i wyświetl go użytkownikowi. Zmiennej `error` przypisz wartość boolowską `true`.
8. Dodaj obiekt przeznaczony do śledzenia operacji utworzenia danych formularza i dodaj wartości do obiektu. W tym celu przeprowadź iterację przez wszystkie dane wejściowe, definiując właściwość o nazwie odpowiadającej nazwie danych wejściowych i wartości odpowiadającej wartości danych wejściowych.
9. Przed końcem funkcji odpowiedzialnej za weryfikację sprawdź, czy błąd nadal istnieje. Jeżeli nie, należy wysłać obiekt formularza.

Prosty quiz matematyczny

W tym projekcie zajmiemy się utworzeniem quizu matematycznego, w którym użytkownik musi odpowiadać na pytania matematyczne. Aplikacja sprawdza odpowiedzi i ocenia ich poprawność. Jako punkt wyjścia możesz wykorzystać zamieszczony tutaj szablon HTML.

```
<!doctype html>
<html>
<head>
  <title>Pełny kurs JavaScriptu</title>
</head>
<body>
  <span class="val1"></span> <span>+</span> <span class="val2"></span> = <span>
  <input type="text" name="answer"></span><button>Sprawdź</button>
  <div class="output"></div>
</body>
</html>
```

Wykonaj wymienione tutaj kroki:

1. W JavaScripcie opakuj kod funkcją `app()`. W tej funkcji utwórz obiekty zmiennych przechowujących wszystkie elementy strony, aby można było ich używać w skrypcie. Ponadto utwórz pusty obiekt o nazwie `game`.

2. Dodaj komponent nasłuchujący zdarzeń `DOMContentLoaded` wywołujący procedurę inicjowania aplikacji po zakończeniu wczytywania strony.
3. W funkcji `init()` do przycisku dodaj komponent nasłuchujący zdarzeń `click` i śledź to zdarzenie w funkcji `checker()`. Ponadto w funkcji `init()` powinna zostać wczytana funkcja `loadQuestion()`.
4. Utwórz funkcję wczytującą pytania oraz kolejną, generującą losowo wybraną liczbę z przedziału określonego przez argumenty tej funkcji.
5. W funkcji `loadQuestion()` wygeneruj dwie losowo wybrane wartości i dodaj je do obiektu `game`. Oblicz sumę tych wartości i tę wartość również przypisz obiektowi `game`.
6. Przypisz wartość i uaktualnij właściwość `textContent` elementów strony, które wymagają dynamicznych wartości liczbowych dla wybranego pytania.
7. Po kliknięciu przycisku należy użyć operatora trójkowego w celu ustalenia, czy udzielono na pytanie poprawnej odpowiedzi. W przypadku takiej odpowiedzi należy wybrać kolor zielony, dla niepoprawnej czerwony.
8. Utwórz element strony wyświetlający wszystkie pytania i śledzone wyniki. W funkcji `checker()` należy dodać nowy element do HTML-a razem ze stylem definiującym kolor w zależności od poprawności odpowiedzi. Wyświetl obie liczby, wynik ich dodawania, a także (w nawiasie) odpowiedź udzieloną przez użytkownika.
9. Usuń zawartość pola danych wejściowych i wczytaj następne pytanie.

Sprawdzian umiejętności

1. Co przedstawione tutaj wyrażenie regularne dopasuje w podanym ciągu tekstowym?

```
Wyrażenie regularne / ([a-e])\w+/g
"Mamy nadzieję, że lubisz JavaScript"
```

2. Czy ciasteczka są częścią obiektu dokumentu?
3. Co przedstawiony tutaj fragment kodu zrobi z ciasteczkami JavaScriptu?

```
const mydate = new Date();
mydate.setTime(mydate.getTime() - 1);
document.cookie = "username=; expires=" + mydate.toGMTString();
```

4. Jakie dane wyjściowe zostaną wygenerowane w konsoli przez ten fragment kodu?

```
const a = "Witaj, świecie!";
(function () {
  const a = "JavaScript";
})();
console.log(a);
```

5. Jakie dane wyjściowe zostaną wygenerowane w konsoli przez ten fragment kodu?

```
<script>
"use strict";
myFun();
console.log(a);
function myFun() {
  a = "Witaj, świecie!";
}
</script>
```

6. Jakie dane wyjściowe zostaną wygenerowane w konsoli przez ten fragment kodu?

```
console.log("a");
setTimeout(() => {
  console.log("b");
}, 0);
console.log("c");
```

Podsumowanie

W rozdziale zostały poruszone pewne tematy zaawansowane, które są ważne, ale prawdopodobnie byłyby za trudne do omówienia we wcześniejszej części książki. Dzięki lekturze tego rozdziału znacznie lepiej rozumiesz wiele obszarów JavaScriptu, a przede wszystkim wyrażenia regularne. Dzięki nim można definiować wzorce ciągów tekstowych i wykorzystywać je do wyszukiwania innych ciągów tekstowych dopasowanych do tych wzorców.

Poruszyliśmy także temat funkcji i obiektu `arguments`, za pomocą którego dostęp do argumentów może odbywać się przy użyciu indeksów. Omówione zostały także hoisting w JavaScripcie i tryb ścisły, który nakłada kilka dodatkowych reguł na JavaScript. Przystosowanie się do pracy z JavaScriptem w trybie ścisłym jest, ogólnie rzecz biorąc, dobrą praktyką i zarazem doskonałym przygotowaniem do pracy z frameworkami JavaScriptu.

Omówione zostało także debugowanie i dostosowanie go do własnych potrzeb. Punkty przerwania i rejestrowanie danych w konsoli pozwalają sprawdzić, co dzieje się w kodzie. Z kolei obsługa błędów pomaga uniknąć nieprzewidywanych awarii programu. Zajęliśmy się również tworzeniem ciasteczek i lokalnego magazynu danych, używaniem JSON-a oraz składnią przekazywania danych. Pokazaliśmy różne rodzaje par klucz-wartość oraz wyjaśniliśmy, jak przetwarzać kod JSON. Omówiliśmy także przechowywanie par klucz-wartość w obiekcie `localStorage` obiektu `window`.

Materiał zamieszczony w rozdziale pomógł w dokładniejszym zrozumieniu JavaScriptu. Zaprezentowaliśmy kilka nowych możliwości nowoczesnego JavaScriptu, choć jednocześnie wyjaśniliśmy, jak radzić sobie ze starszym kodem. W następnym rozdziale zagłębimy się w jeszcze bardziej zaawansowane zagadnienie: współbieżność. Ten temat wiąże się z wielozadaniowością kodu JavaScript.

Skorowidz

C

D

E

F

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

JavaScript: proste narzędzie do złożonych zadań!

JavaScript jest niewielkim językiem skryptowym o imponujących możliwościach. Można w nim tworzyć wiele różnych rodzajów oprogramowania. Doskonale się nadaje do tworzenia dynamicznych stron internetowych, a także do budowy aplikacji internetowych i gier. Mimo upływu lat JavaScript jest niezwykle popularny, a pracę w tym języku ułatwia szereg frameworków, narzędzi i bibliotek. Przyszli zawodowi programiści powinni jednak zacząć naukę od samodzielnego pisania kodu — nawet jeśli na początku wydaje się to trudne.

Dzięki temu przyjaznemu przewodnikowi poznasz kluczowe koncepcje programistyczne i operacje obiektowego modelu dokumentu. Nauczysz się też pisać kod działający asynchronicznie i współbieżnie. Poszczególne zagadnienia zostały zilustrowane przykładowymi fragmentami kodu i prostymi projektami — pozwoli Ci to natychmiast wypróbować działanie tworzonych programów, które w przyszłości mogą posłużyć jako moduły większych aplikacji. Zagadnienia dotyczące JavaScriptu uzupełniono wprowadzeniem do HTML i CSS, co pomoże Ci dokładnie zrozumieć sposób działania nowoczesnych aplikacji internetowych. Przygotujesz się także do pracy z bibliotekami, frameworkami i takimi narzędziami jak React, Angular i Node.js.

W książce:

- konstrukcje logiczne w kodzie źródłowym
- pętle, funkcje i metody JavaScriptu
- współdziałanie z HTML5, współbieżność i programowanie asynchroniczne
- wyrażenia regularne
- wprowadzenie do najważniejszych bibliotek, frameworków i API

Laurence Lars Svekis

tworzy aplikacje internetowe od 1999 roku. Od 2015 roku jest cenionym instruktorem programowania. Jego pasją są nowe technologie.

Maaike van Putten

jest programistką i instruktorką. Od lat pisze w językach JavaScript, Java i Python. Jako specjalistka w dziedzinie programowania uczestniczy w różnych projektach.

Rob Percival

jest programistą i instruktorem w Udemy. Z kursów, które przygotowuje, korzystają miliony użytkowników.

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-197-7

9 788383 221977

Cena: 99,00 zł

Packt