

8. Podstawy JavaScript



Wyzwania:

- Poznasz podstawowe zagadnienia z JavaScript
- Nauczysz się zapisywać proste algorytmy w JS
- Napiszesz własną, prostą grę!

Ocena kursu

Stale podnosimy jakość naszych szkoleń. W związku z tym bylibyśmy wdzięczni za wypełnienie krótkiej ankiety. Zajmie Ci ona dosłownie chwilę.



Ocena bootcampa WebDeveloper

***Wymagane**

W której edycji bierzesz udział? *

Wybierz ▼

Jaka jest Twoja dotychczasowa ocena bootcampa? *

1 2 3 4 5 6 7 8 9 10

Mega słaby ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ Świetny!

Twoja opinia na temat kursu (opcjonalnie)

Twoja odpowiedź

Imię i nazwisko/lub mail jakim logujesz się do bootcampa *

Twoja odpowiedź

PRZEŚLIJ

Nigdy nie podawaj w Formularzach Google swoich haseł.

Formularze Google

Ten formularz został utworzony w domenie Codemy S.A..



Dziękujemy!



8.1. Czym jest JavaScript?



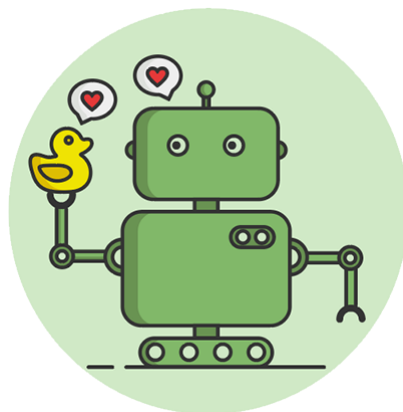
Zaczynamy naszą przygodę z JavaScriptem! Twoje projekty, do tej pory statyczne i reagujące co najwyżej na ruch kursorem, będą mogły nabrać życia i zaoferować użytkownikom nowe funkcjonalności!

Wprowadzenie

Profesor uniwersytetu z pewnością mógłby udzielić Ci bardzo głębokiej odpowiedzi na pytanie "czym jest JavaScript?", ale my nie będziemy Cię zanudzać genezą JSa i zawiłą terminologią. Szczególnie, że zapewne nie możesz się już doczekać, kiedy zaczniesz pisać własne skrypty! ;)

Musimy jednak odpowiedzieć sobie na pytanie: czym *dla nas* jest JavaScript?

Metaforycznie, jest to taki "robot" żyjący w przeglądarce. Za pomocą JavaScriptu możesz mu wytłumaczyć co ma robić. Może wykonywać skomplikowane operacje na liczbach, "na żywo" zmieniać kod HTML na stronie, czy łączyć się z serwerem w internecie, aby pobrać z niego jakieś informacje i wyświetlić użytkownikowi.



Podobnie jak **Metoda gumowej kaczuszki**, ta metafora może wydawać się dziecinna, ale jest bardzo przydatna. Pozwala lepiej zrozumieć, że nasz kod JS to nic innego, jak kolejne "rozkazy" dla tego "roboty".



Możesz spodziewać się ich znacznie więcej w tym module. Wszystkie będą tworzyć większą całość, czyli pozwolą nam na stopniowe rozszerzanie tej analogii o kolejne elementy. Dzięki temu łatwiej będzie Ci poszerzać wiedzę o nowe zagadnienia. Dla ułatwienia i zwiększenia czytelności, kolejne będą zawarte w ramkach z nagłówkiem "Metafora".

Myślenie algorytmiczne

Zanim dotkniemy pierwszej linijki kodu JS, musimy wyjaśnić kwestię myślenia algorytmicznego. Skrypty JavaScript to nic innego, jak zapis algorytmu w pewnym konkretnym języku. O ile łatwo możesz znaleźć mnóstwo informacji na temat samego języka, o tyle trudniej jest nauczyć się myślenia algorytmicznego. Najpierw jednak musimy zrozumieć, czym jest algorytm.

Definicja algorytmu

Algorytm — skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań. Sposób postępowania prowadzący do rozwiązania problemu.

Źródło: [Wikipedia](#)

Z tej definicji wynika, że:

- algorytm musi mieć pewną (skończoną) ilość kroków,
- kroki muszą być jasno zdefiniowane,
- cel algorytmu to wykonanie jakiegoś zadania.

Od razu możemy zwrócić uwagę, że jest to bardzo ogólna definicja. Np. nie wyklucza ona, że algorytm może się składać z algorytmów... Ale do tego tematu jeszcze wrócimy! ;)

Przykłady i ćwiczenia w tym rozdziale mogą wydawać Ci się śmieszne. Pozwalają one jednak na wytrenowanie u siebie umiejętności, które znacznie ułatwią pisanie skryptów. Gorąco zachęcamy do podjęcia próby zmierzenia się z tym wyzwaniem!

Do ilustrowania algorytmów będziemy używać **schematów blokowych**. Nie musisz jednak uczyć się, jak je odczytywać — będziemy tworzyć je krok po kroku, więc bez problemu domyślisz się, co oznaczają.



Zacznijmy od pokazania algorytmu na przykładzie zadania "zrobić herbatę". Możesz pomyśleć, że przecież każdy to wie, i powiedzieć "idź do kuchni, zagotuj wodę i zalej herbatę".

PROSTY ALGORYTM



Jeśli jednak wyobrazisz sobie, że tłumaczysz to samo zadanie dziecku, które nigdy wcześniej nie robiło herbaty, schemat ten może być bardziej szczegółowy.

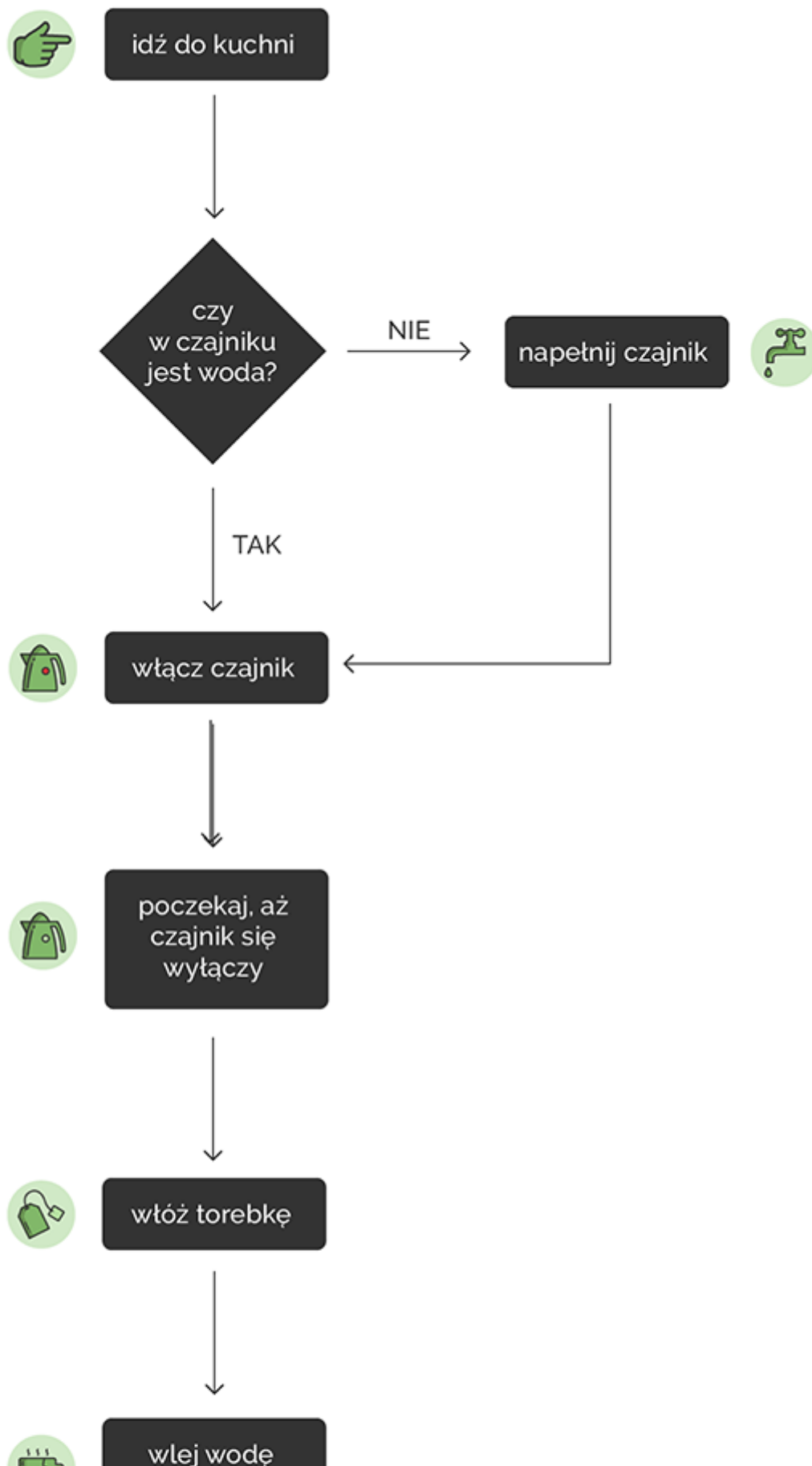
ROZBUDOWANY ALGORYTM





Teraz już mamy przedstawione wszystkie kroki, ale czasem w algorytmie występują też momenty (**decyzje**), w których ścieżka algorytmu się rozdwaja. Dodajmy w takim razie sprawdzenie, czy w czajniku jest woda.

ALGORYTM Z DECYZJĄ

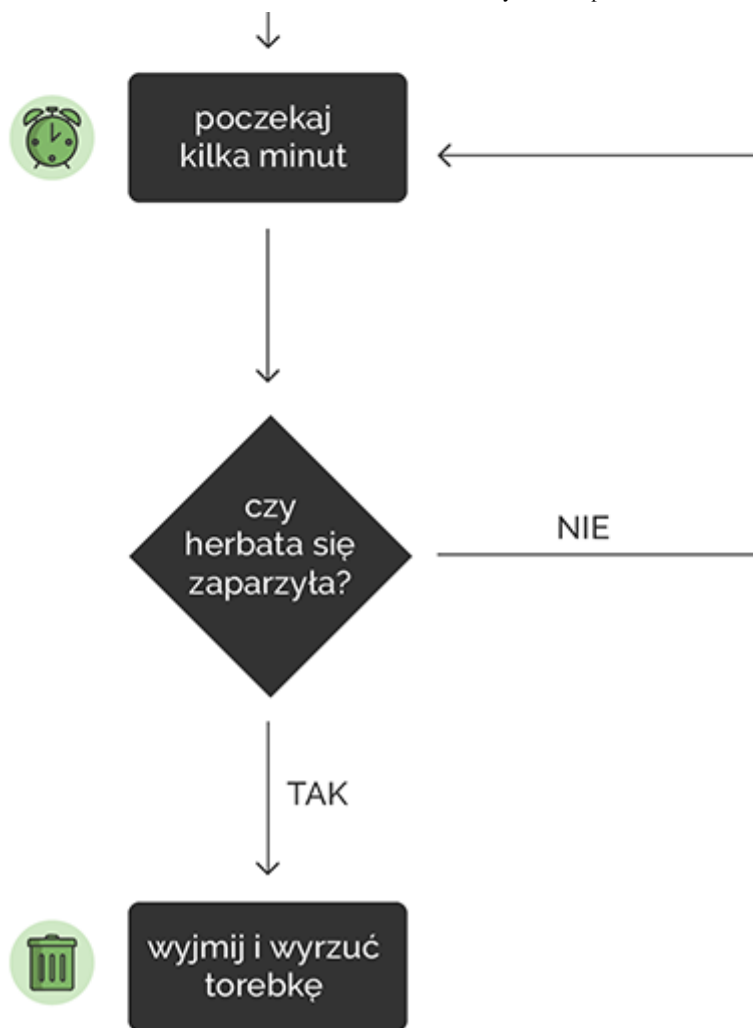




A co się stanie, kiedy spróbujemy to samo podejście zastosować dla sprawdzania, czy herbata już się zaparzyła? Wtedy również zastosujemy decyzję, ale jedna z odpowiedzi będzie nas kierowała do wcześniejszego kroku. W ten sposób otrzymamy **pętlę**, w której będziemy krążyć tak długo, aż zostanie spełniony warunek.

ALGORYTM Z PĘTLĄ



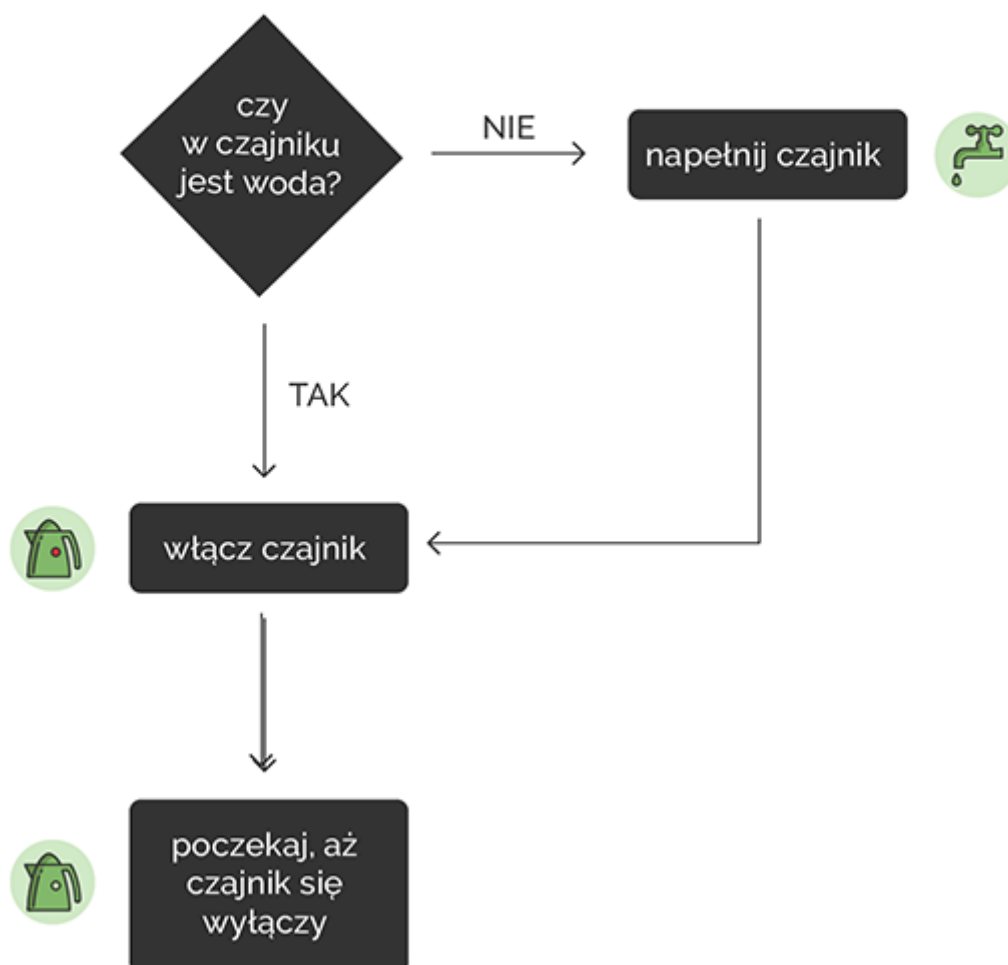


Teraz nasz algorytm zrobił się już nieco skomplikowany. Dlatego dobrze by było wydzielić z niego **procedury**, które będą osobno zdefiniowane. Dzięki temu możemy bardziej skomplikowane elementy zapisać w uproszczonej formie, a szczegóły takiej procedury opisać osobno. Zobaczmy to na przykładzie:

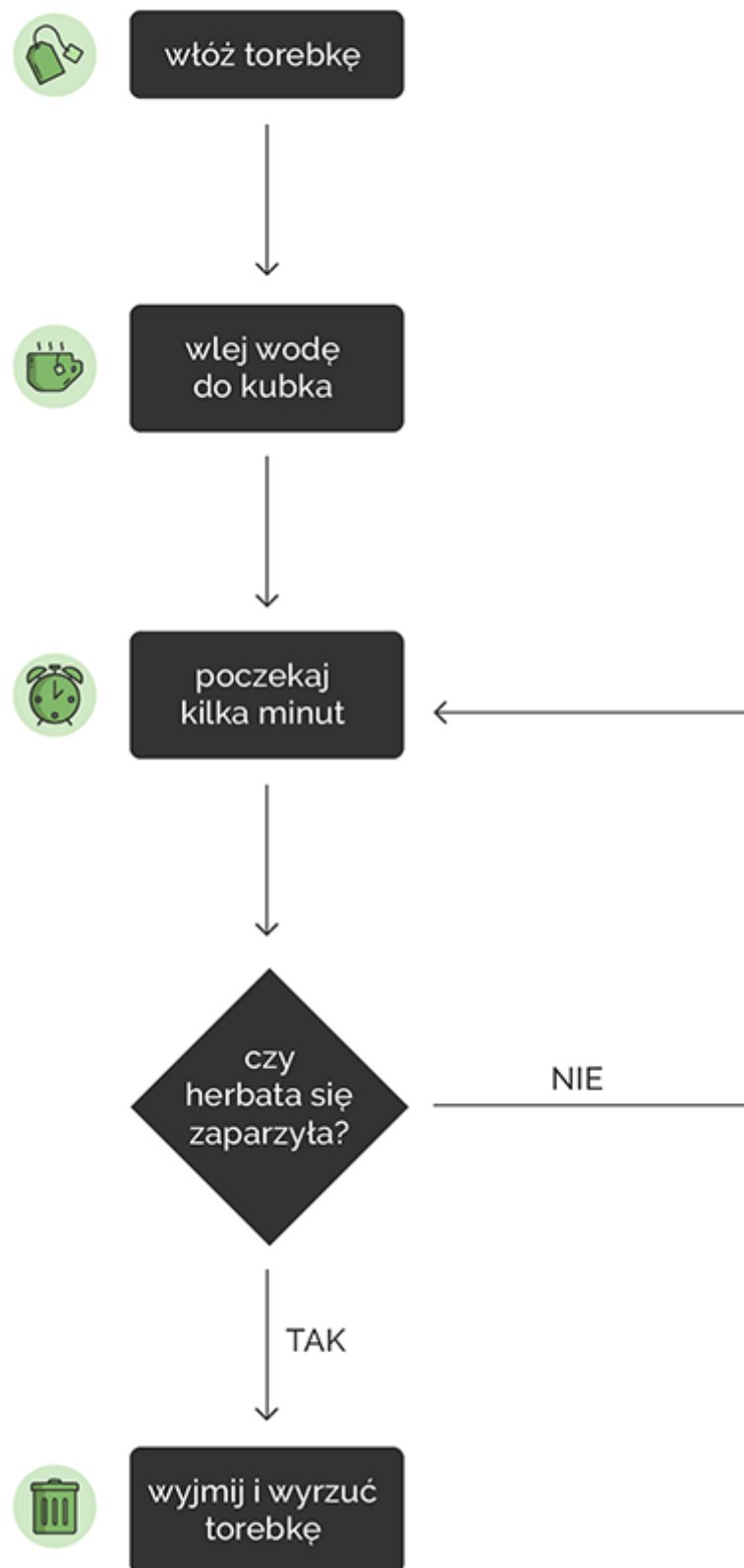
ALGORYTM Z PROCEDURAMI



PROCEDURA "ZAGOTUJ WODĘ"



PROCEDURA "ZALEJ HERBATĘ"



Teraz widzisz, że nasz główny algorytm nie różni się bardzo od pierwotnego, prostego przykładu. Dzięki temu łatwo jest zrozumieć cały algorytm na poziomie ogólnym, a dopiero potem zagłębić się w szczegóły.

Po co nam algorytmy?

Jak wspomnieliśmy na początku rozdziału, skrypt jest algorytmem zapisanym w języku JavaScript. Dlatego warto trenować u siebie myślenie algorytmiczne.

Spróbuj raz dziennie znaleźć jakiś przykład algorytmu w codziennym życiu i rozbić go na kolejne kroki, zauważyć w nim decyzje, pętle i procedury. Możesz rozpisać je za pomocą schematów blokowych, albo po prostu przemyśleć algorytm pod tym kątem. Warto jednak przez kilka tygodni wykonywać takie ćwiczenie, np. w drodze do pracy.

Jak już pewnie się domyślasz, w JavaScriptcie będziemy bardzo często wykorzystywać to, co w algorytmach nazwaliśmy decyzjami, pętlami i procedurami. Jednak ich wykorzystanie będzie wymagało myślenia algorytmicznego, w tym rozumienia problemu na różnych poziomach — od ogólnego, który można naraz "ogarnąć" umysłem i zaplanować potrzebne procedury, aż po szczegółowe, które pozwolą na wypełnienie procedur kodem realizującym ich zadania.

Nie bój się JSa!

W podejściu akademickim zwykle zaczyna się od wyjaśniania pojęć, a dopiero później ich wykorzystanie w praktyce. Nasze doświadczenie pokazuje, że taka droga jest o wiele dłuższa i z tego względu możliwie szybko zaczynamy praktykę, a potem uzupełniamy wiedzę teoretyczną.

Oznacza to, że początkowo nie będziesz rozumieć wszystkiego, co się dzieje w kodzie. Takie jest nasze założenie, więc nie przejmuj się tym. Na początku wystarczy, że będziesz uczyć się jak wykorzystywać JS, nawet jeśli nie wszystko rozumiesz.

Mamy nadzieję, że takie podejście oswoi Cię również z pracą w nieznanym środowisku. Często w pracy web developera możesz spotkać się z sytuacją, kiedy nie musisz rozumieć wszystkiego, by znaleźć i zmienić kluczowy fragment kodu.

W połączeniu z myśleniem algorytmicznym, to podejście pozwoli Ci szybciej adaptować się do nowego środowiska pracy i uczyć się w trakcie pracy, a nie tylko z materiałów edukacyjnych.



Dlatego właśnie używamy w tym module zagnieżdżonych edytorów kodu, które pozwolą Ci bezstresowo eksperymentować!

Zagnieżdżone edytory kodu

W tym module często będziemy używać zagnieżdżonego edytora kodu [CodePen](#). Składa on się z dwóch części:

1. Pierwsza część to edytor kodu. Możesz w nim przełączać się pomiędzy kodem HTML, SCSS i JS.
2. Druga część to podgląd rezultatu kodu z edytora. Podgląd odświeży się automatycznie, kiedy wprowadzisz jakiegokolwiek zmiany w kodzie przykładu.

W kodzie JS będą znajdować się **komentarze**, które są częścią kursu. Dzięki temu będzie nam łatwiej tłumaczyć każdą linię kodu. Nie jest to jednak tylko sposób na naukę — również przy pisaniu własnych skryptów warto pisać komentarze. Dzięki temu:

- wracając do skryptu po jakimś czasie łatwiej będzie Ci zrozumieć co się w nim dzieje,
- w czasie nauki możesz w ten sposób pisać swoje notatki,
- przyzwyczaisz się do czytania kodu innych programistów — najlepsi zawsze komentują swój kod, aby inni programiści mogli bez problemu z niego korzystać.

Zmiany w zagnieżdżonym edytorze nie są zapisywane

Jeśli zmienisz kod w zagnieżdżonym edytorze, zobaczysz jego rezultat w podglądzie. Pamiętaj jednak, że są to zmiany tymczasowe i nie są nigdzie zapisywane.

Dzięki temu możesz od razu eksperymentować z kodem przykładu, nie przejmując się, czy coś się zepsuje — jeśli tak się stanie, wystarczy że odświeżysz stronę, aby ponownie wczytać nasz przykład.

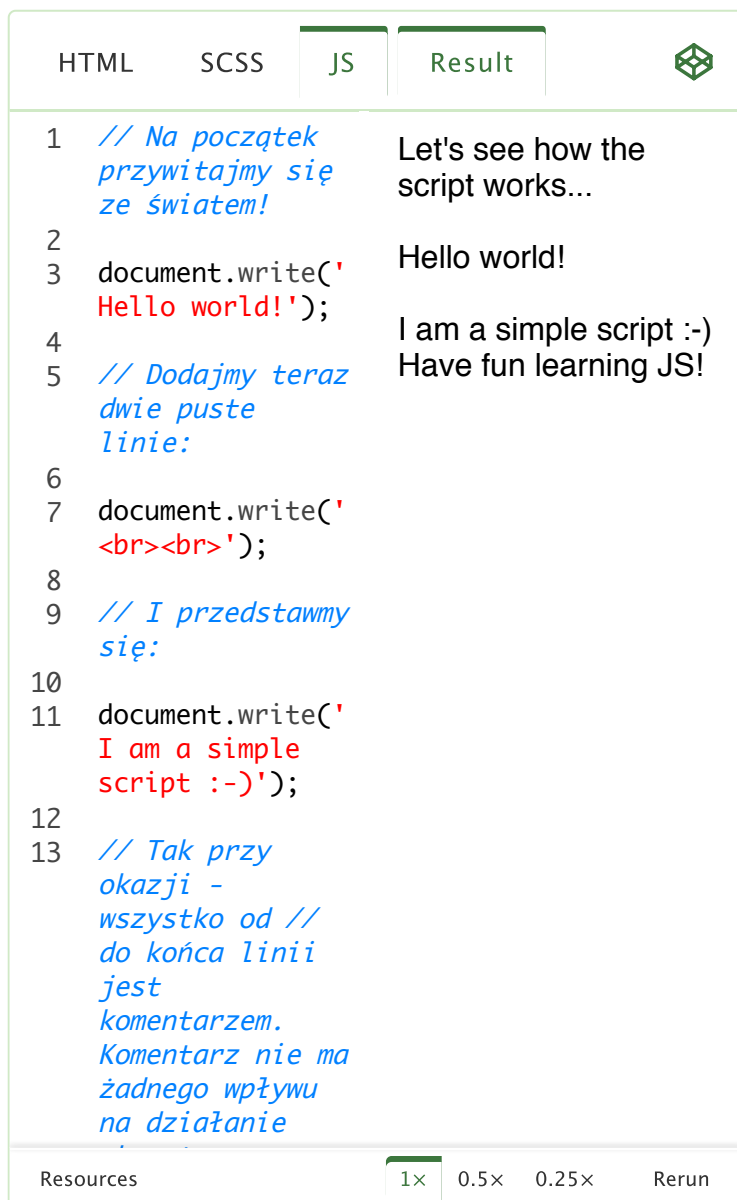
Jeśli zechcesz *zapisać swoje zmiany*, kliknij w logo CodePen w rogu edytora. Dzięki temu będziesz mieć możliwość zapisania własnej wersji. Z tego względu warto założyć konto w serwisie [CodePen](#), aby mieć później dostęp do swoich przykładów.

W zagnieżdżonych edytorach znajdziesz też zadania. Nie są one obowiązkowe i nie będą przez nikogo sprawdzane, ale warto je wykonywać, żeby przetestować swoje rozumienie nowych zagadnień. Zachęcamy też, aby nie kopiować i wklejać innych fragmentów kodu, tylko pisać całe rozwiązanie samodzielnie. Dzięki temu dużo lepiej zapamiętasz przerabiany materiał!



Pierwszy skrypt - hello world!

Zaczynamy od bardzo prostego skryptu, który wypisze na stronie kilka tekstów.



The screenshot shows a code editor with four tabs: HTML, SCSS, JS, and Result. The JS tab is active, displaying 13 lines of JavaScript code. The Result tab shows the output of the script. The code includes comments in Polish and two calls to `document.write()` that output text to the browser. The output in the Result tab shows the text written by the script, including line breaks.

```
1 // Na początek
   przywitajmy się
   ze światem!
2
3 document.write('
   Hello world!');
4
5 // Dodajmy teraz
   dwie puste
   linie:
6
7 document.write('
   <br><br>');
8
9 // I przedstawmy
   się:
10
11 document.write('
   I am a simple
   script :-)');
12
13 // Tak przy
   okazji -
   wszystko od //
   do końca linii
   jest
   komentarzem.
   Komentarz nie ma
   żadnego wpływu
   na działanie
```

Let's see how the script works...

Hello world!

I am a simple script :-)
Have fun learning JS!

Resources 1x 0.5x 0.25x Rerun

Zwróć uwagę, że:

- kod JS jest wykonywany po kolei, od góry do dołu,
- teksty zamykamy w cudzysłowach **'pojedynczych'** lub **"podwójnych"** (zależnie od preferencji — my wolimy pojedyncze).

Dobra praktyka — średniki

Zauważ, że na końcu każdej linii kodu (poza komentarzami) wpisaliśmy średnik. Spróbuj go usunąć — kod będzie działał dalej tak samo. W takim razie, po co używamy średników?

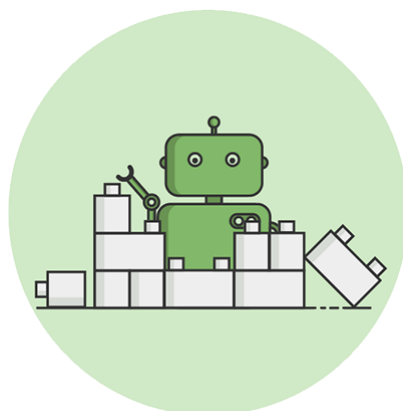


Jest to tzw. dobra praktyka, która pomaga ustrzec się przed pomyłkami. W ogromnym skrócie, średnik na końcu linii jest odpowiednikiem kropki na końcu zdania. Kropka często nie jest konieczna i wszystko byłoby bez niej zrozumiałe, ale czasami może to prowadzić do sporych nieporozumień, kiedy nie wiadomo, gdzie kończy się jedno zdanie, a zaczyna kolejne.

Na razie zwracaj uwagę, aby stosować średniki na końcu każdej linii. Będziemy wskazywać wyjątki, kiedy będą pojawiać się przy omawianych tematach.

Nie musisz wszystkiego rozumieć!

Pamiętaj, że to absolutnie normalne, że nie rozumiesz wszystkiego, co się dzieje w kodzie. Warto przyzwyczaić się do tego, że niektórych rzeczy trzeba się po prostu domyślać.



W powyższym przykładzie skryptu nie wytłumaczyliśmy, jak działa `document.write` i dlaczego tekst do wyświetlenia wstawiamy w nawiasach. Nie jest to jednak dla Ciebie przeszkodą w tym, żeby zmienić wyświetlany tekst, prawda?

Jako web developer będziesz często spotykać się z sytuacjami, kiedy nie rozumiesz każdego elementu kodu. Ważne, żeby domyślać się, co ten kod robi, i jak można go dopasować do swoich potrzeb.

Dlatego nie przejmuj się — z czasem wyjaśnimy większość z tych *niedopowiedzeń*, ale na razie zależy nam na tym, aby jak najszybciej zaznajomić Cię z JSem i pozwolić Ci eksperymentować.

Samodzielne eksperymenty to świetny sposób na naukę — pamiętaj, aby zmieniać coś w każdym przykładzie kodu, żeby lepiej zrozumieć, jak działa.



Z tego względu w tym module przedstawiamy tylko niezbędne podstawy i tłumaczymy je bezpośrednio w zagnieżdżonych edytorach. Wiemy jednak, że w trakcie eksperymentowania może Ci się przydać ściągą - dlatego przygotowaliśmy też dla Ciebie [JS Cheatsheet](#). Dzięki niemu będziesz mieć pod ręką najważniejsze informacje o JS w skondensowanej formie.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✓ Zapoznałem się!

8.2. Zmienne, operacje i warunki



Pierwszy przykład kodu już za nami. Na razie umiesz po kolei wypisywać teksty na stronie. Oczywiście to dopiero początek.

Zmienne

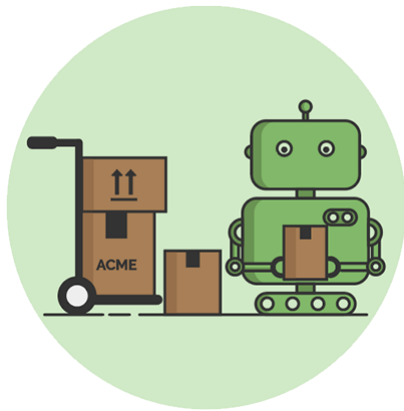
Kolejnym krokiem są **zmienne**. Dzięki nim będziemy mogli "zapamiętać" jakąś wartość, aby później ją wykorzystywać i modyfikować.

Metafora — zmienna

Zmienną możesz wyobrazić sobie jako *pudełko z etykietą*. Na etykiecie jest napisana nazwa tego pudełka.

Pamiętasz, że JS porównaliśmy do robota, któremu wydajemy polecenia? Nie chcemy, żeby się pogubił, i dlatego każde pudełko będzie miało inną etykietę.





Później dowiesz się, że pudełka w różnych pokojach mogą mieć takie same etykiety, ale na razie się tym nie przejmuj.

Pudełko może być na początku puste, ale zwykle będziemy w nim chcieli coś przechowywać. Dla pudełka nie ma różnicy co w nim będzie, tzn. nie mamy np. innych pudełek na liczby, niż na teksty. Możemy też dowolnie zmieniać to co jest w pudełku.

Pamiętaj, że na samym początku musisz złożyć pudełko (zdefiniować zmienną) i napisać coś na etykiecie (nadać nazwę zmiennej).

Deklaracje, przypisywanie wartości i sumowanie liczb

Czas zobaczyć zmienne w akcji! Zaczniemy od prostego działania matematycznego — dodawania.



HTML	SCSS	JS	Result
<pre>1 // Zaczynamy od zadeklarowania zmiennej. Robimy to za pomocą słowa var 2 3 var age = 20; 4 5 document.write(' Defined: age = ' + age + '
'); 6 7 // Każdą zmienną musimy zadeklarować, ale tylko raz! 8 9 var years = 5; 10 11 document.write(' Defined: years = ' + years + '
'); 12 13 // Jeśli potem chcemy zmienić jej wartość, nie używamy już słowa var 14</pre>			<p>Output from the script:</p> <p>Defined: age = 20 Defined: years = 5 Changed value: years = 7</p> <p>Sum of age and years: newAge = 27 Adding years to newAge: newAge = 34 Adding years to newAge: newAge = 41 Increasing newAge by 1: newAge = 42 Increasing newAge by 3: newAge = 45</p>
Resources			1x 0.5x 0.25x Rerun

Używanie 'use strict'

Możesz teraz zadać pytanie: "A co się stanie, jeśli złamię zasady deklarowania zmiennych?". Wyjaśnimy to szerzej, omawiając zakres zmiennych w funkcjach.

Na razie wprowadzimy dobrą praktykę, którą jest dodanie na początku skryptu linii:

```
'use strict'
```

Sprawi ona między innymi, że próba użycia zmiennej, która nie została uprzednio zadeklarowana, spowoduje błąd i zatrzyma działanie skryptu.

Spójrz na ten przykład kodu, który zawiera błąd:

```
var someNumber = 7;  
someNumber = someNumber + 3;  
document.write(someNumber);  
// na stronie wyświetli się liczba 7
```

Oczywiście, kiedy mamy trzy linijki kodu, błąd łatwo znaleźć. Kiedy skrypt ma trzy *tysiące* linii kodu, mamy nieco większy problem. Niejeden programista stracił całe godziny na znalezienie jednej literówki...

Dodawanie vs. łączenie tekstów

Kiedy przyjrzyj się uważnie poprzedniemu ćwiczeniu, to zauważysz, że znaku `+` użyliśmy do dwóch celów: do sumowania liczb (operacji matematycznej) i do wstawienia tekstu (zmiennej). Spójrzmy jeszcze raz, jak to działa.

Dodawanie liczb:

```
var number = 7 + 3;  
document.write(number);  
// na stronie wyświetli się liczba 10
```

Dodawanie tekstów:

```
var animalType = 'cat';  
document.write('I have a ' + animalType);  
// na stronie wyświetli się tekst "I have a cat"
```

Jak widzisz, znak `+` użyty na liczbach sumuje je, a użyty na tekstach po prostu wkleja jeden fragment tekstu za drugim. Co się jednak stanie, gdy będziemy chcieli wykorzystać oba przypadki w jednej funkcji? Spójrz na poniższy kod:

HTML SCSS JS Result

```
1 'use strict';
2
3 // Na początku
  zadeklarujemy
  sobie kilka
  zmiennych:
4
5 var age = 20;
6 var years = 7;
7
8 // Teraz w nowej
  zmiennej
  zapiszemy
  wiadomość i
  wyświetlimy ją
  na stronie.
9
10 var message =
  'After ' + years
  + ' years I will
  be ' + age +
  years + ' years
  old.'
11 document.write('
  First attempt: '
  + message +
  '<br>');
12
13 /* Ups...
  wygląda na to,
```

Output from the script:

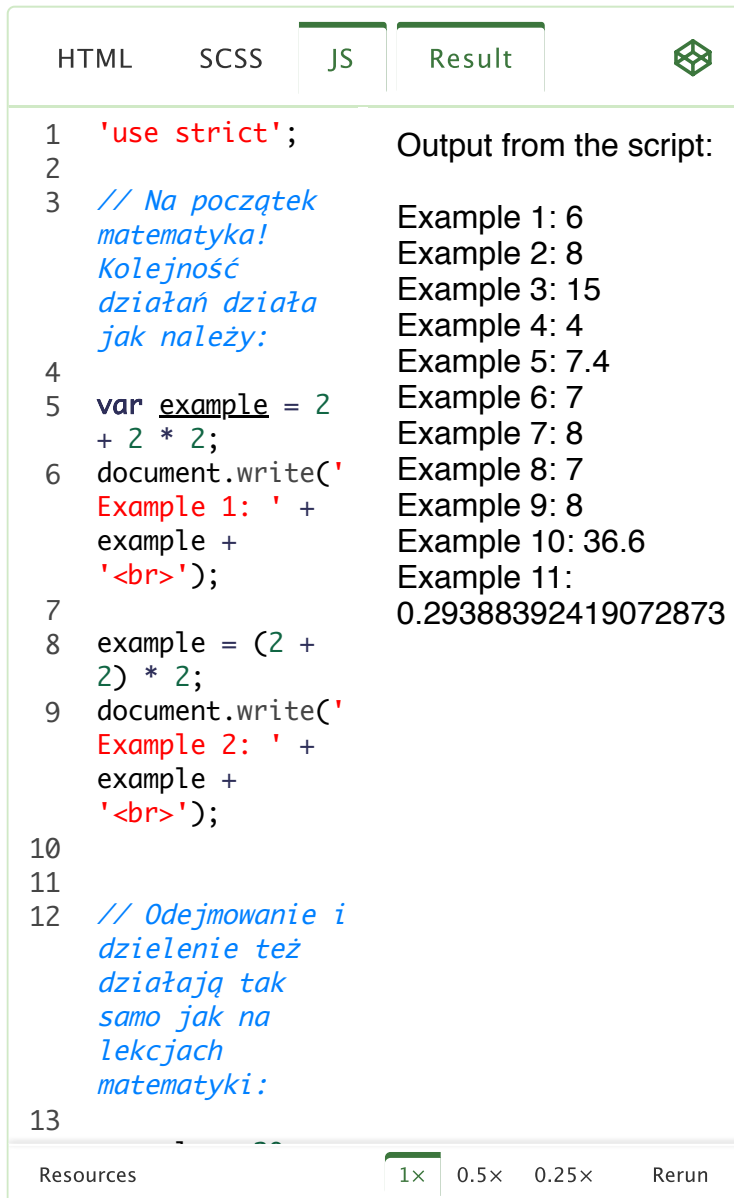
First attempt: After 7 years I will be 207 years old.
With newAge: After 7 years I will be 27 years old.
With brackets: After 7 years I will be 27 years old.

Resources 1x 0.5x 0.25x Rerun

Inne operacje matematyczne na zmiennych

Do tej pory skupiliśmy się na dodawaniu (i łączeniu tekstów), aby wyjaśnić kilka ważnych zagadnień. Teraz szybko pokażemy inne, najczęściej używane operacje na zmiennych. Nie musisz się ich uczyć na pamięć – w razie potrzeby przypomnisz sobie, że *coś takiego było*, i sprawdzisz ponownie poniższy przykład (lub znajdziesz w internecie).





The screenshot shows a web-based JavaScript editor with tabs for HTML, SCSS, JS, and Result. The JS tab is active, displaying the following code:

```
1 'use strict';
2
3 // Na początek
  matematyka!
  Kolejność
  działań działa
  jak należy:
4
5 var example = 2
  + 2 * 2;
6 document.write('
  Example 1: ' +
  example +
  '<br>');
7
8 example = (2 +
  2) * 2;
9 document.write('
  Example 2: ' +
  example +
  '<br>');
10
11
12 // Odejmowanie i
  dzielenie też
  działają tak
  samo jak na
  lekcjach
  matematyki:
13
```

The Result tab shows the output of the script:

Output from the script:

```
Example 1: 6
Example 2: 8
Example 3: 15
Example 4: 4
Example 5: 7.4
Example 6: 7
Example 7: 8
Example 8: 7
Example 9: 8
Example 10: 36.6
Example 11:
0.29388392419072873
```

At the bottom, there are zoom controls (1x, 0.5x, 0.25x) and a Rerun button.

Jest wiele innych operacji matematycznych, ale przedstawiliśmy Ci te najczęściej wykorzystywane. Inne, takie jak np. sinus, bez trudu znajdziesz w [dokumentacji MDN](#).

Inne operacje tekstowe na zmiennych (string)

Na razie będziemy wykorzystywać jedynie łączenie tekstów za pomocą znaku `+`, dlatego nie będziemy teraz rozpraszać Cię innymi operacjami na tekstach. Jeśli jakiegolwiek będą Ci potrzebne do zrozumienia działania przykładu lub realizacji zadań, wymienimy je. Jeżeli w czasie własnych eksperymentów zechcesz np. zbadać długość tekstu lub zmienić w nim jakieś słowo, bez problemu znajdziesz przykłady w internecie, np. w dokumentacji MDN.

Typy zmiennych

Znasz już dwa typy zmiennych:

- **number** czyli liczby,
- **string** czyli teksty.

Niedługo poznasz też wartości prawda/fałsz (**boolean**) oraz funkcje (**function**), a w kolejnym module również tablice (**array**) i obiekty (**object**).

Teraz za to zajmiemy się nieco *dziwniejszymi* wartościami i typami zmiennych: **undefined**, **NaN** oraz **null**.

Pamiętaj jednak, że nie musisz uczyć się tych typów zmiennych na pamięć. Przedstawiamy Ci je po to, aby nie zaskoczyła Cię sytuacja, kiedy Twoje własne eksperymenty wyświetlą jedną z tych wartości. Dodatkowo, wartość typu **undefined** będzie dla nas niedługo ważna.

HTML	SCSS	JS	Result
<pre>1 'use strict'; 2 3 // Zanim zaczniemy przeglądnąć "dziwnych" typów zmiennych, zaczniemy od tych zwykleszych, np. liczb: 4 5 var example = 99; 6 document.write(' Example 1: ' + example + '
'); 7 8 // Możemy sprawdzić typ za pomocą poniższego wyrażenia: 9 10 document.write(' Example 2: ' + example + ', type: ' + typeof(example)</pre>			<p>Output from the script:</p> <p>Example 1: 99 Example 2: 99, type: number Example 3: abcdef, type: string Example 4: undefined, type: undefined Example 5: NaN, type: number Example 6: NaN, type: number, isNaN: true Example 7: true, type: boolean Example 8: null, type: object</p>

Typy zmiennych mogły być dla Ciebie nieco nudnym tematem, bo jeszcze nie wiesz, czego może to się przydać. Ale teraz już wracamy do ciekawszych tematów, które pozwolą nam wspólnie napisać grę na końcu tego modułu!



If/else

Czy pamiętasz jeszcze, jak na początku tego modułu pokazaliśmy myślenie algorytmiczne za pomocą schematów blokowych? Wspominaliśmy wtedy o "decyzjach", kiedy ścieżka schematu blokowego rozdwajała się w zależności od odpowiedzi na zadane pytanie (np. "czy w czajniku jest woda?"). Struktura **if/else** jest właśnie JavaScriptowym zapisem "decyzji" i "rozdwojenia ścieżki".


Metafora - if/else

Strukturę if/else możemy wyobrazić sobie jako sortownię poczty, która w zależności np. od rozmiaru czy wagi przesyłek kieruje je do różnych ciężarówek. W tej sortowni jest jednak zasada, że można zadawać tylko pytania, na które odpowiedź to "tak" lub "nie". Na przykład:

- Czy paczka ma naklejkę "ostrożnie — szkło"? Jeśli tak, musi jechać ciężarówką A.
- Czy paczka jest standardowych rozmiarów? Jeśli tak, musi jechać ciężarówką B.
- Czy paczka waży ponad 10 kg? Jeśli tak, musi jechać ciężarówką C.
- W przeciwnym wypadku musi jechać ciężarówką D.

Składnię if/else najlepiej będzie wytłumaczyć od razu w kodzie.



HTML SCSS JS Result 

```
1 'use strict';
2
3 /* Podstawowa
   składnia
   struktury
   if/else (a
   właściwie "if",
   bo na razie
   jeszcze bez
   "else") jest
   taka:
4
5 if(warunek) {
6     // tu
       wpisujemy kod,
       który będzie
       wykonywany
       tylko jeżeli
       warunek jest
       PRAWDZIWY
7 }
8
9 Zwróć uwagę, że
   w tym przypadku
   nie stawiamy
   średników po
   nawiasach
   klamrowych
   (zarówno po
   otwierającym,
```

Output from the script:

Example 1: condition is TRUE
Example 8: Good afternoon, it is 5 pm.

Resources 1x 0.5x 0.25x Rerun

Dobra praktyka — wcięcia

Zwróć uwagę, że w naszych przykładach zawsze kod wewnątrz nawiasów klamrowych `{ }` ma większe wcięcia niż reszta kodu. To nie jest przypadek, ale kolejny przykład dobrej praktyki, która pomoże Ci uchronić się przed potencjalnymi błędami.

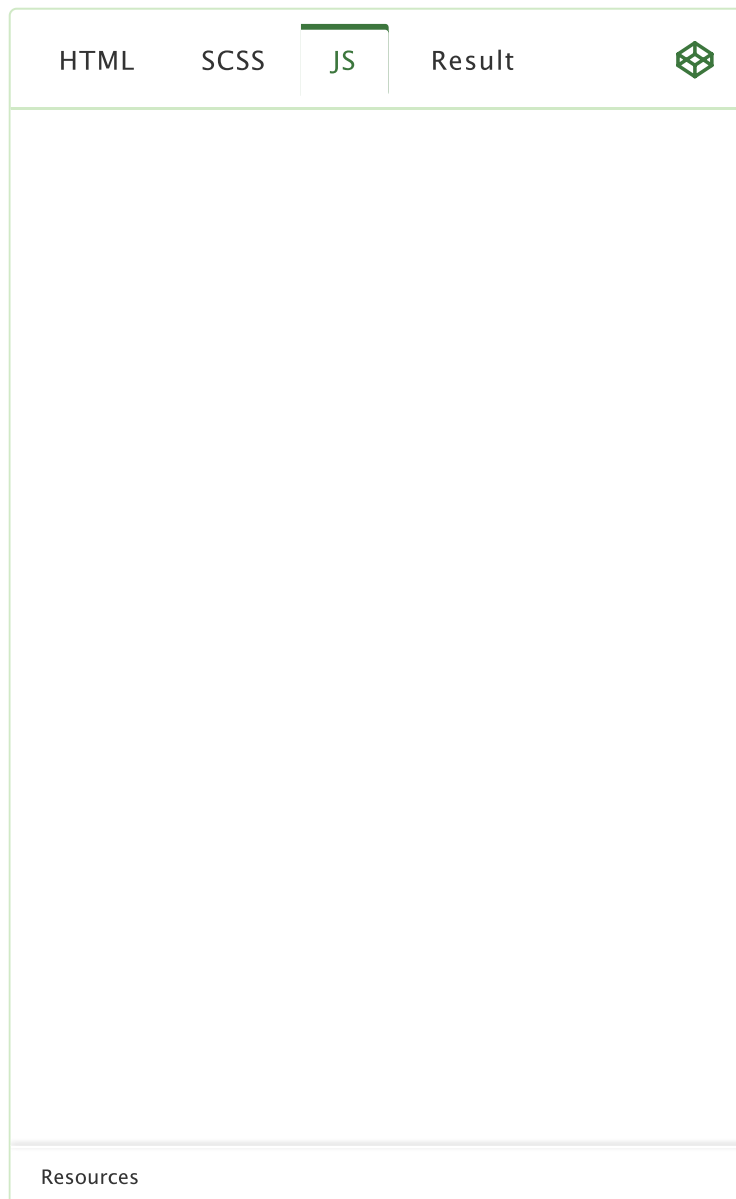
Kod sformatowany w ten sposób jest znacznie bardziej czytelny. Nie trzeba wzrokiem szukać zamykającego nawiasu `}`, żeby wiedzieć, gdzie kończy się blok kodu wykonywanego warunkowo.

Każdy programista musi dbać o porządek w swoim kodzie. W to wliczają się wcięcia w kodzie. Nierzadko zdarza się, że brak poprawnych wcięć od razu dyskwalifikuje kandydata do pracy!

Dbaj o to, aby od początku stosować poprawne wcięcia w kodzie. Dzięki temu wejdzie Ci to w nawyk, nauka będzie szła szybciej, współpraca z mentorem sprawniej, i nie będziesz zmagać się w przyszłości z oduczeniem się złych nawyków.



Wiesz już, w jaki sposób wykorzystywać strukturę if/else, ale na razie umiesz za jej pomocą jedynie porównywać liczby. Teraz skupimy się na tym, co innego można wykorzystać jako warunek.



Poniżej znajdziesz podpowiedzi, które obiecaliśmy w kodzie JS powyższego przykładu.

Pokaż podpowiedź #1

Coś fajnego!

Do tej pory ten moduł może Ci się wydawać dość żmudny. Jest sporo nowych tematów do przyswojenia, a efektem naszych skryptów są tylko teksty napisane na stronie. Te podstawy były Ci jednak potrzebne do tego, aby móc zacząć robić coś ciekawszego.



Teraz wyprzedzimy trochę materiał i użyjemy paru zagadnień z kolejnych submodułów. Na razie nie musisz rozumieć jak i dlaczego działają — pozwolą nam one jednak na stworzenie pierwszego interaktywnego skryptu!

To dalej będzie prosty skrypt, ale jego efekt będzie o wiele ciekawszy. Dzięki temu zobaczysz w akcji wszystko, o czym mówiliśmy do tej pory.

A więc, do dzieła!



Zadanie: Przelicznik temperatury

Czas na pierwsze zadanie z JSa, które sprawdzi Twój mentor!



Wykonaj je w CodePenie — zacznij od wykonania forka (czyli kopii) powyższego przykładu z guzikiem "Say hello!". Po wykonaniu zadania wklej poniżej link do swojego rozwiązania.

1. Konwerter temperatury

Twoim zadaniem jest zrobienie strony, która:

- po kliknięciu guzika zapyta o temperaturę w stopniach Celsjusza,
- jeśli została wpisana liczba, skrypt wyświetli podaną temperaturę oraz jej odpowiednik w stopniach Fahrenheita,
- w przeciwnym wypadku wyświetli informację, że nie wpisano poprawnej liczby.

Wzór na przeliczanie stopni C na F

Temperaturę w stopniach Celsjusza należy pomnożyć przez **1.8**, a następnie do wyniku dodać **32**.

2. Wyświetlanie informacji

Drugą częścią zadania jest podawanie różnych informacji, w zależności od temperatury. Mają być co najmniej trzy warianty, np.:

- czy w tej temperaturze woda jest w stanie ciekłym, czy może jest zamrożona albo w formie pary?
- czy w tej temperaturze nosi się krótkie spodenki, lekką kurtkę czy czapkę i szalik?
- albo wybierz jakieś inne **ciekawe fakty** dotyczące różnych temperatur.

3. Dodajemy drugi guzik

Ostatnia część zadania — dodajemy drugi guzik, który będzie robił dokładnie to samo co pierwszy, ale w drugą stronę. Czyli: będzie pytał o temperaturę w stopniach Fahrenheita, a w odpowiedzi podawać temperaturę w stopniach Fahrenheita, Celsjusza oraz informację z punktu 2.

Powodzenia!

Podgląd zadania

<https://codepen.io/0na/>

Wyślij link ✓



8.3. Funkcje



Podobnie jak przy strukturze if/else, ponownie odwołamy się do rozdziału, w którym tłumaczyliśmy myślenie algorytmiczne. Wydzieliliśmy sobie wtedy procedury, które były osobno opisane. Funkcje są właśnie takimi procedurami.

Co więcej, tak jak wydzielenie procedur w schemacie blokowym pozwalało na bardziej przejrzysty zapis głównego algorytmu, tak też funkcje pozwolą nam na pisanie kodu łatwiejszego w zrozumieniu.

Czym jest funkcja?

Funkcja to fragment kodu, który może być wielokrotnie wykorzystywany. Dzięki temu nie trzeba duplikować tego samego kodu.

Używaliśmy już w pewien sposób funkcji. Pod koniec poprzedniego modułu używaliśmy kodu, który był wykonywany po każdym kliknięciu guzika. Ten kod był właśnie w funkcji, chociaż wykorzystanej w nieco bardziej zaawansowany sposób, który poznasz w kolejnym submodule.

Funkcja może:

- coś przyjmować — funkcja często potrzebuje jakichś informacji, w oparciu o które będzie działać, np. funkcja `document.write` musi dostać tekst, który ma zostać wyświetlony.
- coś robić — funkcja wykonuje pewne operacje, które mogą (ale nie muszą) mieć wpływ na cokolwiek poza funkcją, np. funkcja `document.write` powoduje wyświetlenie tekstu na stronie,
- coś zwracać — funkcja może wstawiać jakąś "odповідź" w miejsce jej wywołania, np. funkcja `window.prompt` zwraca odpowiedź wpisaną przez użytkownika.

Zwróć uwagę, że różne funkcje będą posiadać różne kombinacje z powyższych punktów, czyli np. jakaś funkcja może coś przyjmować i coś wykonywać, ale niczego nie zwracać. Dokładnie tak się dzieje w przypadku funkcji `document.write`.



Możesz sobie wyobrazić funkcję, jako maszynę. Są różne maszyny — np. pralki, windy, czy nawet samoloty. W wielu przypadkach nie wiesz, jak taka maszyna działa, i nie musisz tego wiedzieć, aby z niej korzystać. W przypadku naszej metafory będzie tak samo w niektórych przypadkach, zaś w innych będziemy sami budować naszą maszynę.

Weźmy na przykład pralkę. Musisz coś do niej włożyć (brudne ubrania, proszek), bo bez tego jej działanie nie ma sensu. Pralka potem coś robi przez jakiś czas, a na końcu dostajemy z powrotem czyste ubrania.

W przypadku funkcji będziemy dostarczać informacje (liczby, teksty, etc.), które nazywamy argumentami, a w rezultacie funkcja zwróci nam coś innego (inne liczby, inne teksty).

Innym rodzajem maszyny jest winda. Tu nie musimy niczego dostarczyć, ani niczego nie dostaniemy w zamian. Za to winda wykona dla nas jakąś pracę.

Istnieją także funkcje, którym nie podajemy argumentów i które niczego nie zwracają. Będą one jednak wykonywać jakąś operację (np. zmieniać kolor tła na stronie).

Najważniejsze jest jednak, że tej maszyny możemy używać wielokrotnie. Niektóre maszyny zachowują się za każdym razem tak samo, inne — w zależności od tego co im dostarczymy. Tak czy inaczej, nie musimy budować nowej pralki do każdego prania.


Pamiętaj też, że samo kupienie pralki nie sprawi, że Twoje ubrania będą czyste! Podobnie i funkcja musi być stworzona (zadeklarowana), ale robi coś dopiero kiedy ją uruchomimy (wywołamy).

Niektóre funkcje, z których będziemy korzystać, są wbudowane w przeglądarkę. Oznacza to, że zostały już zadeklarowane gdzieś w kodzie Twojej przeglądarki i nie trzeba ich deklarować ponownie. Często jednak będziemy pisać własne funkcje, aby nie wklejać wielokrotnie tego samego kodu.

Pamiętaj, że samo zadeklarowanie funkcji jest tylko zapisaniem algorytmu, który ma być przez nią wykonany. Dopiero po wywołaniu tej funkcji algorytm zostanie wykonany.

Zobaczmy jak to działa w praktyce:



HTML SCSS JS Result 

```
1 'use strict';
2
3 /* Zaczniemy od
   napisania
   bardzo prostej
   funkcji. Jak na
   pewno
   pamiętasz, do
   wyświetlania
   jakiegoś tekstu
   na stronie
   często
   używaliśmy tej
   funkcji:
4 document.write(
   'Hello world!
   <br>');
5
6 Zadeklarujemy
   teraz funkcję,
   która będzie
   robiła
   dokładnie to co
   powyżej -
   pisała "Hello
   world!" i
   dodawała znak
   nowej linii.
   Funkcję
   będziemy
```

Output from the script:
Hello world!
My name is JavaScript
:)
I am a programming
language.
5 squared is 25
7 squared is 49
3
2
1
0

Resources 1x 0.5x 0.25x Rerun

Dobra praktyka — camelCase

W powyższym przykładzie zaczęliśmy używać nieco bardziej złożonych nazw zmiennych. Zapisane zostały w stylu, który nazywa się *camelCase*. Jest to standard nazywania zmiennych w JavaScriptcie.

Zasada jest bardzo prosta: w nazwie zmiennej nie można używać spacji, myślników czy kropek (o wyjątkach dowiesz się niedługo). Chcemy jednak, żeby nazwy zmiennych były czytelne.

Jeśli chcemy użyć kilku słów w nazwie zmiennej, każde z nich (poza pierwszym!) zaczynamy wielką literą. Oczywiście, staramy się nie przesadzać z ich długością...

```
var theNumberWeGetFromMultiplyingTheNumberFiveByItself = 5 * 5;
```

Ale tak przesadzony przykład dobrze pokazuje o ile bardziej czytelny jest zapis *camelCase* od używania wyłącznie małych liter:



```
var thenumberwegetfrommultiplyingthenumberfivebyitself = 5 * 5;
```

... i krótszy od używania podkreślników:

```
var the_number_we_get_from_multiplying_the_number_five_by_itself = 5
```

Zakres zmiennych

Do tej pory nazywaliśmy zmienne jak tylko chcieliśmy. Oczywiście domyślamy się, że nie można nazwać zmiennej słowem `function`, `var` czy `return`, bo są to szczególne słowa, które mają swoje znaczenie w JavaScriptcie.

Warto jednak się zastanowić, jak to się dzieje, że zespoły programistów piszą strony z tysiącami linii kodu JS, a jednak nie mają problemu z tym, że dwóch z nich użyje tej samej nazwy zmiennej. W końcu można by się spodziewać, że wtedy kod jednego programisty będzie nadpisywał wartości zmiennej drugiego programisty, prawda?

Tu właśnie z pomocą przychodzi nam zakres zmiennych. To pojęcie mówi właśnie o tym, w jakiej "przestrzeni" zmienna istnieje. Otóż zmienna istnieje tylko wewnątrz funkcji, w której została zadeklarowana.

Metafora — zakres zmiennych

Przypomnijmy sobie wcześniejsze metafory: - JavaScript to robot, który wykonuje nasze polecenia, - zmienna to pudełko z etykietą, - funkcja to maszyna, np. pralka.

Wyobraźmy sobie teraz fabrykę z przyszłości (taką z filmów science-fiction), która jest w pełni zautomatyzowana. Można by powiedzieć, że taka fabryka również jest maszyną.

Materiały do produkcji są dostarczane do fabryki w pudełkach. Na wjeździe do fabryki każde pudełko jest rozpakowane i zapakowane w pudełko z logo naszej fabryki. Otrzymuje też zupełnie nową etykietę identyfikacyjną, która działa tylko wewnątrz tej fabryki, i to po niej będziemy rozpoznawać to pudełko.

Co więcej, nasza fabryka z tych materiałów produkuje komponenty, które dopiero później będą składane w gotowy produkt. Te komponenty również są pakowane w pudełka z naszymi etykietami.



Pudełka, których używamy, są jednak brzydkie i szare, a my bardzo dbamy o nasz wizerunek, dlatego nie mogą one nigdy opuścić naszej fabryki. Każde zamówienie jest pakowane w jeden kontener i w nim wywożone z fabryki.

Metaforycznie, właśnie tak działa zakres zmiennych.

Zarówno nazwy argumentów, jak i zmienne zadeklarowane wewnątrz funkcji, istnieją tylko wewnątrz tej funkcji. Co więcej, te argumenty i zmienne są jednorazowe, tzn. przestają istnieć po zakończeniu wywołania funkcji. Kolejne wywołanie funkcji już ich nie pamięta.

Pomyśl o tym, jak wywoływalismy funkcję `log` z poprzedniego przykładu — za każdym razem, kiedy ją wywołaliśmy, argument `text` wewnątrz tej funkcji miał nową wartość. Właśnie dlatego w przykładzie z guzikiem "Say hello!" musieliśmy zadeklarować zmienną `name` poza funkcją wykonywaną po kliknięciu guzika, aby dało się zrobić ostatnie ćwiczenie w tym przykładzie. Wróć teraz do niego i zwróć uwagę na to, jaki jest zakres zmiennej `name`.

Funkcja ma też dostęp do wszystkich zmiennych zadeklarowanych poza nią. Na razie jednak nie będziemy tego wykorzystywać — może to prowadzić do wielu nieporozumień, dlatego powinno się korzystać z tej możliwości bardzo ostrożnie.

Zobaczmy teraz na kilku przykładach jak działa zakres zmiennych:



HTMLSCSSJSResult

1'use strict';
2
3// Zaczniemy od
zadeklarowania
zmiennej
globalnej.
Zmienna ta
będzie dostępna
w całym kodzie
JS, nie tylko w
danym pliku.
4
5var
globalVariable
= 'abc';
6
7/* Jak łatwo
się domyślić,
takie podejście
nie jest
najlepsze.
Nazwy zmiennych
mogą się
powtarzać w
różnych plikach
czy fragmentów
kodu. Z tego
względu
powinniśmy dbać
o zmniejszanie

Output from the script:

exampleArgument
printed from inside the
function: lorem ipsum
example1 printed from
inside the function: def
exampleArgument
printed from OUTSIDE
the function: undefined
example1 printed from
OUTSIDE the function:
undefined
globalVariable printed
from inside the
function: ghi
globalVariable printed
from OUTSIDE the
function: ghi
example3 printed from
inside the function: bar
example3 printed from
OUTSIDE the function:
foo
example4 printed from
inside the
encapsulation: It
works!

Resources1x0.5x0.25xRerun

Debugowanie

W tym module używamy zagnieżdżonych edytorów kodu, aby umożliwić Ci eksperymentowanie od razu w momencie czytania modułu. Ma to jednak na celu również przyzwyczajanie Cię do pisania komentarzy, które pomagają zrozumieć kod, oraz do ciągłego sprawdzania, jak działa każdy krok Twojego skryptu.

Przed wszystkim jest to świetny sposób na naukę. Pozwala Ci widzieć na bieżąco wartość każdej zmiennej i każdej zmiany w kodzie. Jest jednak jeszcze jedna zaleta – w przypadku błędu w kodzie zauważysz go bardzo szybko i łatwo będzie Ci zlokalizować, w którym miejscu wkradła się pomyłka.



Z tego względu gorąco zachęcamy do używania `document.write` w zagnieżdżonym edytorze. Już niedługo poznasz też `console.log`, który jest znacznie bardziej przydatny przy standardowej pracy w plikach zapisanych na Twoim komputerze.

W ten czy inny sposób, wyświetlaj informacje o tym, że dana funkcja została uruchomiona, jakie argumenty otrzymała i co jest jej wynikiem. To znacznie pogłębi Twoje rozumienie JavaScriptu i pozwoli Ci o wiele szybciej usamodzielnąć się i tworzyć coraz ciekawsze skrypty.

Zadanie: Użycie funkcji w skrypcie

Wróć do swojego zadania z poprzedniego submodułu, tj. do przelicznika temperatury. Wykonaj fork (kopię) tamtego zadania, aby pracować pod innym adresem niż wcześniej.

Twoim zadaniem jest wydzielenie fragmentów kodu z poprzedniego zadania i zapisanie ich jako funkcji.

1. Funkcja, która przyjmuje temperaturę C i zwraca temperaturę F.
2. Funkcja, która przyjmuje temperaturę C i zwraca informację np. o stanie skupienia wody.
3. Funkcja, która przyjmuje temperaturę F i zwraca temperaturę C.
4. Funkcję, która wyświetla tekst na stronie, odpowiednio dodając nową linię.

Zastosuj te funkcje w kodzie zadania, aby wyeliminować powtarzanie tego samego kodu i zwiększyć czytelność skryptu.

Zwróć uwagę, że funkcję 2 wykorzystasz przy obu guzikach.

Podgląd zadania

<https://codepen.io/0na/>

Wyślij link ✓



8.4. DOM i eventy



JavaScript daje nam możliwość wprowadzania zmian na stronie. Do tej pory zmiany te ograniczały się do wypisywania tekstów na stronie za pomocą `document.write`. Wyjątkiem był przykład, w którym zadawaliśmy pytanie użytkownikowi po kliknięciu guzika. Tam odnosiliśmy się do dwóch elementów: do guzika oraz do diva, w którym wypisywaliśmy teksty.

Zanim jednak nauczymy się, w jaki sposób odwoływać się do elementów na stronie i w różny sposób wpływać na nie, zacznijmy od wyjaśnienia sobie, co mamy na myśli, mówiąc o elementach DOM.

HTML vs. DOM

Być może znasz już te pojęcia i różnice między nimi. W praktyce wiele osób nazywa "HTMLem" zarówno HTML, jak i DOM. Postaramy się jednak wyjaśnić różnicę i trzymać się poprawnego nazewnictwa.

DOM oznacza **Document Object Model**, czyli **Obiektowy model dokumentu**.

W wielkim skrócie, kod HTML jest w pliku `.html`. Kiedy ten plik zostanie otworzony w przeglądarce, na jego podstawie zbuduje się DOM. Dlatego `div` w pliku `.html` jest elementem *HTMLa*, zaś w przeglądarce jest elementem *DOM*.

Kiedy otworzysz narzędzia developerskie w przeglądarce, np. wybierając z menu kontekstowego (po kliknięciu PPM) pozycję "Zbadaj element" (ang. *Inspect element*), zobaczysz coś, co wygląda jak kod HTML, ale w rzeczywistości jest to drzewo elementów DOM, wygenerowane przez przeglądarkę na podstawie kodu HTML.

Możesz zobaczyć różnicę między nimi, jeśli w kodzie HTML masz np. niezamknięty tag `<p>`. Przeglądarka sama spróbuje się domyślić, gdzie on powinien być domknięty. Przeglądarka oczywiście nie zmienia pliku `.html`, tylko odpowiednio generuje DOM.

Spróbuj w pliku `test.html` umieścić **BŁĘDNY** kod:

```
Lorem <p> ipsum <ul> <li> dolor
```



Następnie wyświetl ten plik w przeglądarce i otwórz zbadaj ten element. W narzędziach developerskich zobaczysz, że przeglądarka sama uzupełniła tagi zamykające:

W tym module dowiesz się również, że element DOM ma w sobie o wiele więcej, niż tylko informacje z kodu HTML. Co więcej, kiedy za pomocą JavaScriptu będziemy dodawać, usuwać lub zmieniać jakieś elementy strony, będą to właśnie operacje na DOMie, nie na HTMLu.

Na razie jednak wystarczy zapamiętać, że w pliku `.html` mamy kod HTML, ale już w przeglądarce mamy elementy DOM.

Wybieranie i zmiany elementu DOM

Modyfikacje, które możemy chcieć wprowadzić na stronie za pomocą JSa, będą najczęściej polegać na dodaniu lub usunięciu klasy lub zawartości jakiegoś elementu. Aby było to możliwe, musimy wskazać w JSie, na jakim elemencie chcemy wykonywać te operacje.



HTML SCSS JS Result

```
1 'use strict';
2
3 /* Na początek w
   kodzie HTML
   umieściliśmy
   dziesięć divów z
   klasą "box".
   Każdy z nich ma
   inne id, a
   dodatkowo trzy
   pierwsze mają
   klasę "box-bg",
   która zmienia
   kolor tła.
4
5 Zaczniemy od
   znalezienia
   elementu o
   konkretnym id.
   Służy do tego
   funkcja
   document.getElem
   entById, której
   używaliśmy już w
   przykładzie i
   zadaniu z
   guzikami.
6 */
7
8 var box13 =
```

This box is green!
And it is **beautiful**

#box2.box-bg

#box3.box-bg

#box4

#box5

Resources 1x 0.5x 0.25x Rerun

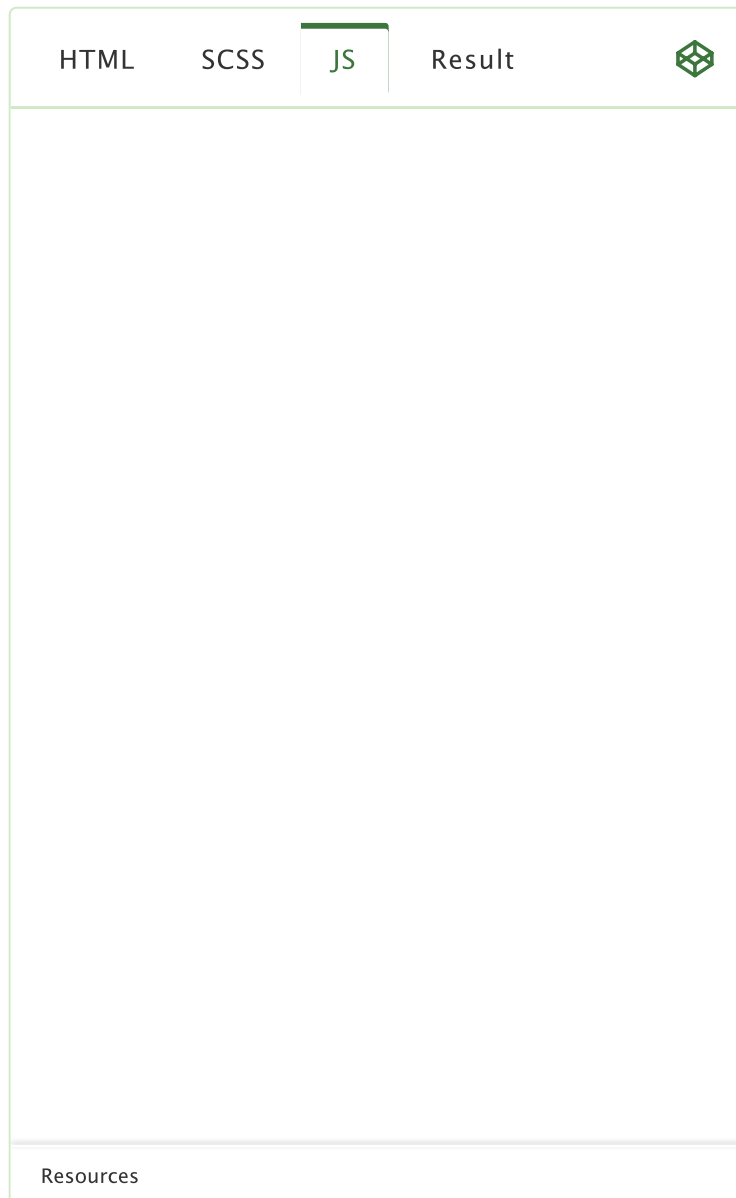
Trawersowanie drzewa DOM

O ile czasami będziemy szukać elementu o konkretnym **id** czy klasie, często będziemy potrzebowali znaleźć element wiedząc tylko o tym, gdzie jest w drzewie DOM względem znanego nam elementu.

Weźmy na przykład wyskakujące okienko z komunikatem o promocji na stronie sklepu internetowego. Po kliknięciu w "x" modal ma zostać ukryty. Za chwilę dowiesz się, jak powiązać akcję ze zdarzeniem kliknięcia, ale na razie skupmy się na tym, że wtedy znamy tylko element, który został kliknięty, czyli "x". Musimy na tej podstawie znaleźć cały element, w którym ten "x" się znajdował.



Właśnie takie przechodzenie po drzewie DOM, czyli korzystanie z relacji rodzic-dziecko, nazywamy trawersowaniem drzewa DOM.



Eventy - wprowadzenie

Umiesz już znajdować element DOM i poruszać się po drzewie DOM. Teraz zostało nam jeszcze jedno zagadnienie, które pozwoli nam tworzyć interaktywne skrypty. Mówimy oczywiście o zdarzeniach, czyli **eventach**.

Przykładem eventu może być **click** — korzystaliśmy już z kodu, który reagował na kliknięcia. Za chwilę poznasz również inne rodzaje zdarzeń.

Metafora — event

Event możesz wyobrazić sobie jako zdarzenie, które każdy może zaobserwować. Przykładem może być zapalenie się zielonego światła na skrzyżowaniu.



To samo zdarzenie będzie miało różne skutki dla różnych osób:

- piesi w ogóle nie patrzą na sygnalizator dla samochodów, więc w żaden sposób nie zareagują na zapalenie się zielonego światła,
- kierowcy jadący na wprost ruszą i przejadą przez skrzyżowanie,
- kierowcy skręcający w prawo będą mogli skręcić, ale muszą najpierw udzielić pierwszeństwa pieszym przechodzącym przez jezdnię, w którą chcą skręcić,
- kierowca karetki jadącej na sygnale nie przejmuje się światłem, ale zielone światło do jazdy na wprost mówi mu, że nie musi się obawiać samochodu wjeżdżającego na skrzyżowanie z innego kierunku.

Zwróć uwagę, że nie każdy na skrzyżowaniu patrzy na to światło, i nie każdy, kto je obserwuje, zareaguje w ten sam sposób.

Do każdego elementu DOM można dodać *event listener*, który będzie nasłuchiwał, czy na tym elemencie miało miejsce zdarzenie konkretnego typu. Kiedy to się wydarzy, zostanie uruchomiona podana przez nas funkcja, zwykle nazywana **callbackiem**. Dzięki temu dla różnych kombinacji elementów i zdarzeń (a nawet wielokrotnie dla tych samych kombinacji) możemy tworzyć różne akcje, wykonywane w odpowiedzi na event.

HTML SCSS JS Result

```
1 'use strict';
2
3 /* Zaczynamy od
   prostego
   przykładu - w
   momencie
   kliknięcia w
   nagłówek
   pierwszego boks
   chcemy włączać i
   wyłączać jego
   specjalne style.
4
5 Najpierw musimy
   znaleźć element,
   na którym chcemy
   nasłuchiwać
   zdarzenia.*/
6
7 var box1Header =
   document.querySelector('#box1
   header');
8
9 /* Następnie
   używamy na nim
   funkcji
   addEventListener
   , która jako
   argumenty
```

First

Hello world!!!
Lorem ipsum

Second

Dolor sit amet.

Third

Lorem ipsum

Resources 1x 0.5x 0.25x Rerun

Magiczne słowo **this**

W powyższym przykładzie musieliśmy osobno definiować funkcje, które wypisują tekst w boksach, ponieważ w każdej z nich używamy odwołania do konkretnego boks.

Tu z pomocą przyjdzie nam słowo **this**, które w każdej funkcji będzie miało nieco inne znaczenie. Na razie interesuje nas to, że w funkcji będącej callbackiem eventu słowo **this** będzie oznaczało "ten element, na którym wychwycono zdarzenie".

To nam znacznie ułatwi pisanie kodu, ponieważ umożliwi wykorzystanie tej samej funkcji dla wszystkich boksów z naszego przykładu.



HTML SCSS JS Result

```
1 'use strict';
2
3 /* Użyjemy
   identycznego
   kodu, jak w
   poprzednim
   przykładzie, z
   tym że
   zamieniliśmy
   "box3" na
   "this".
4 */
5
6 var eventCallbackFor
   AllBoxes =
   function(event){
7
   this.insertAdjacentHTML('beforeEnd', 'Detected
   event ' +
   event.type +
   '<br>');
8 };
9
10 var box1 =
   document.getElementById('box1');
11 var box2 =
```

First

Lorem ipsum

Second

Dolor sit amet.

Third

Lorem ipsum

Resources 1x 0.5x 0.25x Rerun

Propagacja i domyślne akcje

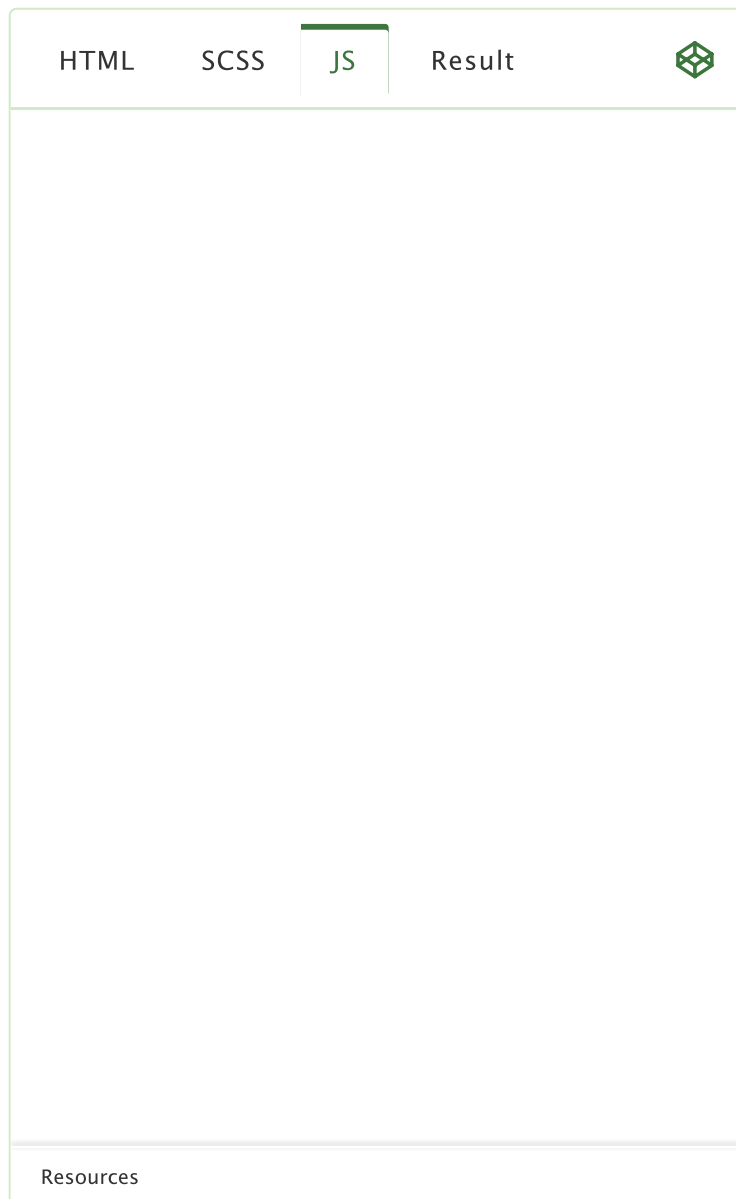
Musimy sobie zadać filozoficznie brzmiące pytanie: w co właściwie klikam?

Jeśli klikniesz w guzik na stronie, to czy jednocześnie nie klikasz w `section`, w którym on się znajduje? I w `div`, w którym znajduje się ta sekcja? I w końcu też w `body` ?

Event zawsze wykonuje się na jakimś konkretnym elemencie DOM, zwanym celem zdarzeniem (*event target*). Następnie ten sam event zostanie wykonany również na rodzicu tego elementu, i jego rodzicu, i jego rodzicu, i tak aż do `body`, `html`, i `document`.



Właśnie dlatego kliknięcie w obrazek, który jest w linku, powoduje przejście do podlinkowanego adresu. A skoro już wspominamy o linkach, wyjaśnimy też sobie, jak sprawić, aby link nie zmieniał adresu strony.



Zadanie: Piszemy własną grę!

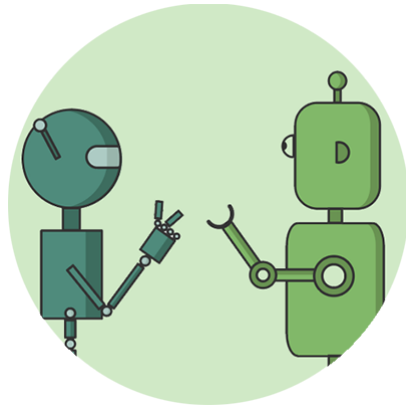
W tym module było sporo nowych informacji, i nie były to banalne tematy. Było to jednak niezbędne minimum, aby umożliwić Ci samodzielne eksperymentowanie i pisanie własnych skryptów.

Teraz Twoim zadaniem jest napisanie strony, która umożliwi grę w papier-kamień-nożyce.

Grę możesz przygotować w CodePenie lub w repozytorium na GitHubie.

Zasady gry

Zapewne znasz zasady tej gry, ale nawet w takiej sytuacji dobrze jest jasno sprecyzować oczekiwany efekt.



Pojedyncza runda gry polega na jednoczesnym (w naszym wypadku – pozornie jednoczesnym) zagraniu dwóch graczy.

Każdy gracz może wybrać jeden z ruchów: papier, kamień, lub nożyce.

W naszym wypadku będzie jeden gracz, a rolę drugiego gracza będzie pełnił skrypt. Potocznie mówi się, że gramy z komputerem, więc tak też będziemy się odnosić do tego wymyślnego przeciwnika.

Wynik rundy może być następujący:

- remis – jeśli obaj gracze wybrali ten sam ruch
- wygrana gracza, który zagrał papier, jeśli drugi gracz zagrał kamień,
- wygrana gracza, który zagrał kamień, jeśli drugi gracz zagrał nożyce,
- wygrana gracza, który zagrał nożyce, jeśli drugi gracz zagrał papier.

Gra toczy się do osiągnięcia przez jednego z graczy wcześniej ustalonej liczby wygranych rund.

Plan realizacji zadania

Poniżej opisujemy kolejne etapy tworzenia aplikacji. Do zaliczenia tego zadania obowiązkowa jest realizacja wszystkich trzech etapów.

Jeśli ta część zadania pójdzie Ci sprawnie, możesz też rozwinąć swoją aplikację wedle własnego pomysłu, np. dodać tabelę wyników, zapamiętywać najdłuższą serię wygranych gracze bez żadnej porażki, etc.

Bez paniki!



Jeżeli na pierwszy rzut oka zadanie wydaje Ci się trudne — nie przejmuj się. Przeczytaj jeszcze raz spokojnie zasady gry i rozrysuj je sobie jako algorytm w postaci **schematu blokowego**.

Następnie, czytając instrukcje do poszczególnych etapów, zauważ, że wszystkie ich składowe (warunki, pętle, losowanie liczb, elementy reagujące na kliknięcie, wpisywanie tekstów w `div` ...) były już pokazywane na przykładach w tym module. Twoje zadanie polega na tym, by krok po kroku złączyć te elementy w jedną całość.

Jeżeli nie pamiętasz, jak napisać daną funkcjonalność, wróć po prostu do odpowiedniego fragmentu w tym module.

Etap 1 — minimalna funkcjonalność

Podstawą naszej gry będą trzy guziki: "papier", "kamień" i "nożyce". Kliknięcie jednego z nich będzie oznaczało ruch gracza.

Na stronie oprócz guzików umieścimy też `div` o `id="output"`, w którym będą pojawiać się komunikaty generowane przez grę.

Do każdego z guzików przywiąż funkcję, która wywoła funkcję `playerMove` z odpowiednim argumentem. Ta funkcja powinna:

- jako argument przyjmować nazwę ruchu gracza,
- losować liczbę z zakresu 1-3 i zapisywać ją w zmiennej,
 - jeśli ta liczba to 1, komputer zagrał papier,
 - jeśli ta liczba to 2, komputer zagrał kamień,
 - jeśli ta liczba to 3, komputer zagrał nożyce,
- w oparciu o wylosowaną liczbę, skrypt decyduje czy wynikiem jest remis, wygrana gracza, czy wygrana komputera,
- rezultat rundy jest wyświetlany w outpucie w formacie "YOU WON: you played PAPER, computer played ROCK".

Do losowania liczby 1-3 oraz do wyświetlania tekstu na stronie napisz osobne funkcje, które wykorzystasz wewnątrz funkcji `playerMove`.

To wszystko! Już taka bazowa wersja gry umożliwi rozgrywanie kolejnych rund!

Pamiętaj, aby gruntownie przetestować, czy wszystko działa jak należy! ;)

Etap 2 — liczenie wygranych rund

Gra w papier-kamień-nożyce zwykle nie kończy się po jednej rundzie. Dodaj w takim razie licznik rund wygranych przez każdą ze stron.



Dodaj osobny `div` o `id="result"`, w którym będzie wyświetlany wynik w formacie "X - Y", gdzie X to wygrane gracza, a Y to wygrane komputera. Remis nie zmienia wyniku.

Pamiętaj, aby przemyśleć miejsce zadeklarowania zmiennych, które będą przechowywać te informacje. Zmieniaj wartość tych zmiennych przy każdym wywołaniu funkcji `playerMove`, w zależności od tego, kto wygrał.

Nie zapomnij też po każdej rundzie na nowo wyświetlić wyniku na stronie!

Etap 3 — nowa gra i liczba rund

Dodajemy nowy guzik — "New game". Po naciśnięciu tego guzika:

- ma pojawić się prompt pytający o liczbę wygranych rund, która kończy grę,
- do czasu zakończenia gry, na stronie ma być widoczna informacja o tym, jaka liczba wygranych rund oznacza zwycięstwo.

Po rundzie, w której gracz lub komputer osiągnął wymaganą liczbę wygranych rund, do komunikatu o wygranej lub przegranej rundzie należy dodać w nowej linii w outpucie komunikat o zakończeniu gry i jej wyniku, np. "YOU WON THE ENTIRE GAME!!!".

Od tego momentu dalsza gra nie powinna być możliwa, a klikanie w guziki "papier", "kamień" i "nożyce" powinno powodować wyłącznie dodanie w nowej linii w outpucie komunikatu "Game over, please press the new game button!".

Pamiętaj, że możesz dla ułatwienia stworzyć sobie np. jakąś zmienną, która ma wartość `true` lub `false`, a od jej wartości zależy, czy dalsza gra jest możliwa, czy nie.

Podsumowanie

Gdybyśmy zaczęli ten moduł od pokazania Ci tej gry, pewnie trudno byłoby Ci uwierzyć, że w tak krótkim czasie nauczysz się samodzielnie napisać tego rodzaju grę.

Zamiast tego zaczęliśmy ten moduł od wytłumaczenia Ci myślenia algorytmicznego. Ćwiczysz codziennie to myślenie, prawda? ;)

Myślenie algorytmiczne jest niezbędne do tego, aby zadanie "napisz grę w papier-kamień-nożyce" rozpiąć krok po kroku, jak powyżej. Jest też niezbędne do tego, aby powyższą instrukcję zamienić w działający kod.

Pamiętaj, aby na każdym kroku wyświetlać na stronie efekt działania ostatniej (albo ostatnich paru) linii kodu. Dzięki temu nawet jeśli popełnisz błąd, szybko go wyśledz. Pomoże Ci to również lepiej rozumieć, co się dzieje w kodzie Twojej aplikacji.



Nie bój się też eksperymentowania. Jeśli nie wiesz jak coś zrobić, spróbuj się domyślić. Ćwicząc tę umiejętność znacznie przyspieszysz zdobywanie doświadczenia i szybciej zaczniesz myśleć jak doświadczony programista!

Powodzenia!

Podgląd zadania

<https://0na.github.io/Pa1>

Wyślij link ✓

[Regulamin](#)

[Polityka prywatności](#)

© 2019 Kodilla

