**9. Wprowadzenie do JavaScriptu część 2**

**Wyzwania:**

* Dowiesz się czym są funkcje i jak z nich korzystać.
* Poznasz konstrukcje tablic oraz ich zalety.
* Stworzysz prostą grę.

**9.1. Funkcje**

Funkcje to podstawowy element każdego języka programowania.

Funkcja wyodrębnia fragment kodu, aby mógł on stać się powtarzalny.

Jeśli robisz coś w skrypcie wielokrotnie - to zamiast kopiować kod jeden pod drugim - należy użyć funkcji. Jest to zgodne z regułą - DRY (ang. Don't Repeat Yourself). Po co pisać coś wiele razy, skoro można tylko raz? :) Funkcje zapewniają takie samo działanie jak przy powielaniu, a kodu jest o wiele mniej i łatwiej nim zarządzać.

Aby móc używać funkcji, trzeba ją najpierw zadeklarować. Każda funkcja powinna posiadać następujące elementy:

* **Function name** - *nazwa* - po nazwie będzie można się do niej odwołać,
* **Parameters** - *parametry* - parametry są jak zmienne z tą różnicą, że funkcja może je otrzymać; są to jedynie nazwy zmiennych, które wykorzystamy, jeśli funkcja jakieś parametry otrzyma,
* **Curly brace / block** - *klamry/blok* - za pomocą klamer należy zdefiniować, kiedy funkcja zaczyna się i gdzie się kończy,
* **Body function** - *ciało funkcji* - kod, który jest zapisany wewnątrz bloku funkcji,
* **Return** - *return* - specjalna instrukcja, która kończy działanie funkcji i zwraca wartość jako wynik funkcji.

**Deklaracja funkcji**

Deklaracja przedstawiona jest na poniższym przykładzie:

function functionName(parameter) {

// Place where the body of the function is located

// Performing operations

return 'returnedValue';

// The functions does not need to return anything, so return can be omitted

}

Tutaj definiujemy, jak funkcja ma działać, kiedy zostanie już wywołana.

Może to jeszcze niezbyt oczywiste, ale wewnątrz funkcji można robić wszystko to, co do tej pory - deklarować zmienne, wykonywać na nich operacje arytmetyczne czy wykorzystywać instrukcje warunkowe.

Funkcja jest po to, aby jakiś kod móc wykonywać wielokrotnie bez konieczności pisania go kilka razy.

Tyle o deklaracji. Przykłady będą jeszcze omówione.

**Wywołanie funkcji**

Deklaracja nie jest tym samym, co wywołanie. Aby wykonać funkcję, należy ją wywołać. Robi się to poprzez zapisanie jej nazwy oraz okrągłych nawiasów. Dopiero poniższe polecenie sprawi, że funkcja zostanie wykonana.

functionName();

**Zwracana wartość**

Wewnątrz funkcji można wyodrębnić fragment kodu, który coś robi. Funkcja może (ale nie musi) zwracać wynik swojego działania.

Jeśli funkcja posiada polecenie return, to znaczy, że coś zwraca, czyli przekazuje swoją wartość końcową do elementu kodu, który ją wywołał.

var functionValue = functionName();

W powyższym przykładzie do zmiennej functionValue przypisujemy to, co funkcja zwróci (czyli w tym przypadku napis “zwracanaWartosc", spójrz na deklarację).

Funkcja może zwrócić dowolny typ danych, który jest dostępny w języku Javascript. Należy pamiętać, że polecenie return przerywa działanie funkcji.

**Parametry funkcji**

Funkcje mogą przyjmować parametry. Dzieje się to tak, że przekazane w wywołaniu parametry będą dostępne pod określoną zmienną, którą można wykorzystać wewnątrz funkcji.

Wyobraźmy sobie grę w kamień, papier, nożyce (w tym momencie nie jest istotne to, w jaki sposób się ta gra toczy) - z pewnością potrzebna jest nam funkcjonalność, która rozstrzygnie, który gracz zwyciężył rundę. Funkcja nada się idealnie, ponieważ nawet przy wielu partiach, jeden fragment kodu sprawdzi nam wszystkie możliwości i zwróci wynik rundy.

var userChecked = 'stone',

computerChecked = 'scissors';

function checkResult (userChoice, computerChoice) {

// Check if it was a draw

// if yes, then return the value 'Draw'

// Make sure the user wins

// if yes, return the value 'User wins'

// Otherwise

// return the value 'Computer wins

}

var winner = checkResult (userChecked, computerChecked);

Na początku zadeklarowane zostały dwie zmienne reprezentujące wybór graczy: (userChecked oraz computerChecked).

Następnie zadeklarowana została funkcja checkResult, która przyjmuje dwa parametry: userChoice i computerChoice (nazwy zmiennych celowo są inne). Deklaracja ta oznacza, że w momencie wywołania funkcji będziemy oczekiwać, że te parametry będą posiadały wartość.

Wewnątrz funkcji zostanie umieszczony kod, który sprawdzi, który z graczy zwyciężył. Oczywiście chodzi o słowo kluczowe return, jednak w tej chwili nie będziemy skupiać się na wyłanianiu zwycięzcy, a przekazaniu i odczytaniu parametrów.

Poniższy kod to wywołanie funkcji z przekazaniem parametrów. Zapis ten sprawia, że kod zawarty w naszej funkcji zostanie wykonany, a przekazane zmienne będą dostępne jako parametry.

checkResult (userChecked, computerChecked);

W tym wypadku parametr funkcji userChoice przyjmie wartość zmiennej userChecked, a computerChoice przyjmie wartość zmiennej computerChecked.

Zwróć uwagę, że parametr nie jest tym samym, czym zmienna. Choć podobieństwa są dosyć oczywiste - obie posiadają wartość, to ich użycie jest całkowicie odmienne, a parametry są dostępne tylko wewnątrz funkcji.

**Dobre praktyki**

Najlepiej, aby jedna funkcja robiła jedną rzecz, nie więcej.

**Co to znaczy?**

Jeśli Twój skrypt ma za zadanie pobrać od użytkownika dane, potem je przeliczyć i dać jakiś wynik, a na sam koniec wyświetlić to użytkownikowi na stronie, to przyda się tutaj zapisać co najmniej 3 osobne funkcje:

* pobranie danych
* policzenie wyniku
* wyświetlenie danych na stronie

Zapytasz: 'A czemu nie jedna ?'. Otóż w jednej również się da i Twój kod może działać zupełnie tak samo, jednak mówimy tutaj o dobrych praktykach.

Funkcje powinny być krótkie - mają robić jedną rzecz, ale za to dobrze. Jeśli funkcja jest krótka, to łatwo ją zrozumieć. Łatwo ją także zmienić/poprawić, jeśli robi tylko jedną rzecz, a nie na przykład trzy.

To sprawdzone podejście.

Polecam Ci zapoznać się z zasadami SOLID.

Poniżej masz link do krótkiego wpisu na ten temat - <https://www.p-programowanie.pl/paradygmaty-programowania/zasady-solid/>

Pamiętaj, że funkcje służą do wykonywania jakiegoś kodu wielokrotnie. Po wywołaniu funkcji spodziewamy się określonego wyniku. Jeśli funkcja robi wiele rzeczy, to możliwość jej wykorzystania np. w innym kontekście staje się trudna.

Warto przyzwyczajać się do takiego podejścia, nawet jeśli twój skrypt nie będzie posiadał dużej funkcjonalności.

**Jak nazywać funkcje**

Nazwa funkcji powinna wyjaśniać, za co dana funkcja odpowiada. Zasada jest taka sama co przy zmiennych, chociaż tutaj można dodatkowo podzielić funkcje na te, które coś zwracają, ustawiają wartości lub coś robią.

Te, które coś ustawiają, na przykład imię gracza, nazywa się z przedrostkiem set. Na przykład setUserName(name).

Te, które coś zwracają, na przykład imię gracza, nazywa się z przedrostkiem get. Na przykład getUserName(), getMatchResult().

Te, które coś robią, na przykład losują wybór komputera, powinniśmy nazwać znacząco, aby od razu było wiadomo, co ta funkcja robi, na przykład randomizeComputerChoice().

**Podsumowanie rozdziału**

Dowiedzieliśmy się, czym są funkcje i do czego mogą się przydać.

Wiesz, dlaczego warto podzielić kod z wykorzystaniem funkcji. Potrafisz stworzyć funkcje bez parametrów oraz z parametrami, a także zwrócić wartość funkcji.

Wiesz również jak powinno się nazywać funkcje i dlaczego warto trzymać się konwencji nazewnictwa.

**Zadanie: Tworzymy funkcje**

1. Stwórz funkcję, która przyjmuje dwa parametry i nazwij ją getTriangleArea - function getTriangleArea(a, h) {}.
2. Za pomocą instrukcji warunkowej sprawdź, czy przekazane parametry a oraz h mają wartość większą od zera. Jeśli któryś z parametrów nie spełni tego warunku (będzie mniejszy bądź równy zero), zwróć wartość 'Nieprawidłowe dane'.
3. Jeśli oba parametry są dodatnie, to poniżej instrukcji warunkowej dodaj polecenie return, aby funkcja zwracała wynik działania a\*h/2.
4. Poniżej deklaracji funkcji wywołaj następujące polecanie: console.log(getTriangleArea(10, 6)). Jeśli wszystko zrobiłeś prawidłowo, w konsoli powinien pojawić się wynik działania.
5. Zrób trzy zmienne, które będą przechowywały pole trójkąta, za każdym razem przekazując różne liczby w parametrze. Podpowiedź: var triangle1Area = getTriangleArea(10, 15);
6. Stwórz nowe repozytorium i wyślij tam swoje zadanie, a link do niego wklej poniżej. Pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/108235)

Początek formularza

 Wyslij link

Dół formularza

**9.2. Tablice**

Tablice zdążyliśmy już poznać.

Potrafisz je stworzyć, nadpisać oraz odczytać ich wartości.

Dobrze wiedzieć, że tablice są obiektem i udostępniają dodatkowe właściwości oraz metody operacji na nich samych.

**length**

To właściwość tablicy, która przedstawia aktualny rozmiar tablicy (ilość elementów w tablicy).

Załóżmy, że mamy taką tablicę:

var array = ['HTML', 5, 'World', 9.99, {name: 'Piotr'}, 99];

console.log(array.length); // It will display 6

Jak widzimy, 'length' zwróci wartość 6. No i to jest prawdą.

Musimy jednak pamiętać, że tablice zaczynają się od indeksu równego 0, a zatem ostatni element ma indeks 5.

**toString()**

Jest to funkcja, która pozwala nam łączyć wszystkie elementy tablicy jako jeden tekst, domyślnie oddzielając poszczególne elementy znakiem ",".

Niestety nie obsługuje ona wypisywania danych z obiektów.

Przykładowy kod:

var array = ['aaa', [2.55, 'bbb', [5] ] ];

var stringWithArray = array.toString();

console.log(stringWithArray); // "aaa, 2.55, bbb, 5"

A co jeżeli będziemy mieli zawarty w środku obiekt?

var array = ['aaa', [2.55, 'bbb', [5, {id: 'hello' } ] ] ];

var stringWithArray = array.toString();

console.log(stringWithArray); // "aaa, 2.55, bbb, 5, [object Object]"

Jak widać, zamiast tekstu hello dostaliśmy [object Object], czyli typ naszej zmiennej.

Aby wypisać dane z obiektu, musielibyśmy posłużyć się pętlą for in, którą poznamy nieco później.

**join()**

Funkcja join(), podobnie jak metoda toString(), funkcja join() pozwala nam na zmianę tablicy na typ *string*. Jednak w tym przypadku mamy możliwość zdefiniowania, jakim znakiem kolejne z wpisów będą od siebie oddzielone. Ta funkcja musi mieć podany parametr, nawet jeśli to będzie prosty *string* - variable.join('').

var test = [ 'aaa', 2.55, 'bbb', 5 ];

var testToString = test.join(' | ');

console.log(testToString); // aaa | 2.55 | bbb | 5

**push()**

Zdarza się, że chcemy dodać nowe dane do istniejącej już tablicy.

Dane dopisywane są zawsze na końcu tablicy.

Możemy posłużyć się taką składnią:

var test = ['a', 'b', 'c'];

test[3] = 'd';

W wyniku tego kodu na końcu naszej tablicy zostanie dodany string 'd'.

Dlaczego więc używamy funkcji push()?

Jest ona bardziej przystosowana do dokładania danych do naszej tablicy, a co za tym idzie znacznie szybsza w działaniu.

Dodatkowo jako wynik działania zwraca długość tablicy.

var test = ['a', 'b', 'c'];

var x = test.push('d');

Tablica będzie wyglądała tak samo jak w poprzednim przykładzie, lecz dodatkowo zmienna x będzie miała wartość 4, czyli nową długość tablicy test.

**pop()**

Jeżeli chcemy usunąć ostatni element z tablicy, posłużymy się funkcją pop(). Można powiedzieć, że ściąga ona ostatni element tablicy oraz zwraca go jako wynik działania.

var test = ['a', 'b', 'c'];

var x = test.pop();

Po tej operacji, tablica będzie wyglądała następująco - ['a', 'b'], natomiast zmienna xzawierać będzie usunięty element, czyli 'c'.

**splice()**

Funkcja splice() pozwala na dodawanie i/lub usuwanie elementów do/z tablicy. W codziennej pracy na tablicach jest, krótko mówiąc, bardzo bardzo użyteczna, więc warto ją znać.

Zobaczmy, jakie parametry przyjmuje funkcja.

var array = ['a', 'b', 'c'];

array.splice(index, elements, new1, new2, … , newX);

Argumenty index i elements są obowiązkowe, więc nie można ich pominąć.

index określa na której pozycji dodamy/usuniemy element.

elements definiuje, ile elementów zostanie usuniętych z tablicy. Jeśli ustawimy go jako 0, żaden z elementów nie zostanie usunięty.

new1 - newX, to lista elementów, które zostaną dodane do tablicy przed elementem array[index].

**Dodawanie elementów**

Przyjmijmy taki kod. Zachęcam do wklejenia go do edytora.

var array = ['a', 'b', 'c'];

array.splice(1, 0, 'x', 'y', 'z');

console.log(array);

Po jego wykonaniu nasza tablica będzie wynosiła: ['a', 'x', 'y', 'z', 'b', 'c']

Co się tutaj dzieje?

Przed elementem array[1], czyli 'b', zostanie usuniętych zero elementów, a następnie wstawione zostaną przekazane elementy 'x', 'y', 'z'.

**Usuwanie elementów**

Sprawdźmy teraz, jak usunąć 2 elementy z naszej tablicy.

var array = ['a', 'x', 'y', 'z', 'b', 'c'];

var x = array.splice(2, 2);

console.log(array); // ['a', 'x', 'b', 'c']

console.log(x); // ['y', 'z']

Tablica wyniesie ['a', 'x', 'b', 'c'], natomiast zmienna x będzie równa wartości ['y', 'z'].

Co się tutaj dzieje?

Począwszy od elementu array[2], czyli 'y', zostanie usuniętych 2 elementów łącznie z tym, na który wskazał *index*, czyli 2.

Wynikiem funkcji będą elementy ściągnięte z tablicy, zostaną one przypisane do wartości x.

**concat()**

Aby połączyć ze sobą tablice, posłużymy się funkcją concat().

Metoda przyjmuje dowolną ilość tablic i zwraca nową, która jest sumą przekazanych tablic.

var test = [1, 2, 3];

var test2 = ['a', 'b', 'c'];

var test3 = ['x', 'y', 'z'];

var arr = test.concat(test2,test3);

console.log(arr); // [1, 2, 3, 'a', 'b', 'c', 'x', 'y', 'z'];

Co się tutaj dzieje?

Do tablicy test dołączamy tablice test2 oraz test3. Wynikiem funkcji jest nowa tablica, która zawiera wszystkie wartości z tablic.

Tablice test, test2, test3 pozostają niezmienione, natomiast wartość zmiennej arrwynosić będzie [1, 2, 3, 'a', 'b', 'c', 'x', 'y', 'z'];.

W praktyce zostaje utworzona nowa tablica o tych samych wartościach, co test i do niej zostaną dołączone pozostałe tablice przekazane jako parametry funkcji concat().

**indexOf()**

Ta funkcja jako parametr przyjmuje wartość szukanego elementu.

Przykładowo chcemy sprawdzić, czy nasza tablica zawiera imię 'Ola'.

var names = ['Asia', 'Kasia', 'Ola', 'Sylwia', 'Ola'];

var indexOfOla = names.indexOf('Ola');

console.log('The search value is on ' + indexOfOla + ' position');

console.log(names[indexOfOla]);

Należy jednak pamiętać, że jeśli szukanych wpisów jest więcej, funkcja zwróci tylko indeks pierwszego napotkanego elementu.

**map()**

Ta funkcja jako parametr przyjmuje funkcję map() działa w taki sposób, że przechodzi po kolei po każdym z elementów tablicy i wywołuje funkcję (podaną jako parametr) na kolejnych elementach tablicy.

Zwracana wartość map() to również tablica, jednak wewnątrz funkcji będącej parametrem można ją zmodyfikować.

var values = [1, 2, 3, 4, 5, 6];

var multipliedValues = values.map(function(value) {

return value \* 10;

});

console.log(values);

console.log(multipliedValues);

Powyższy skrypt zwraca tablicę wartości, w której każdy z jej elementów został pomnożony razy 10 w stosunku do pierwotnej wartości.

Co się tutaj dzieje?

Najpierw zadeklarowana zostaje tablica values.

Następnie deklarujemy zmienną multipliedValues, która będzie zawierać wynik funkcji map wykonanej na tablicy.

Funkcja map() działa podobnie do pętli, z tym że wykona się tyle razy, ile elementów znajduje się w tablicy. Za każdym razem zostanie również wywołana funkcja, która została przekazana jako parametr do funkcji map().

Za pierwszym wykonaniem funkcji parametr value przyjmie pojedynczą wartość tablicy. Ta wartość zostanie pomnożona przez 10, następnie zwrócona jako wynik tej funkcji, a w rezultacie zostanie dodana do tablicy będącej wynikiem funkcji map().

**filter()**

Działa podobnie jak map() i również porusza się po każdym kolejnym elemencie tablicy i wywołuje funkcję będącą parametrem dla każdej wartości tablicy.

Funkcja filter() zwraca tablicę elementów, dla których została zwrócona wartość true w przekazywanej funkcji.

Reszta elementów jest pomijana - najlepiej widać to na poniższym przykładzie:

var names = ['Asia', 'Kasia', 'Ola', 'Sylwia', 'Ola', 'Ania'];

var namesWithoutOla = names.filter(function(name) {

console.log('name in filter: ' + name);

return name != 'Ola';

});

console.log(namesWithoutOla);

Powyższa funkcja filter() zwróci tablicę imion bez imienia 'Ola'. Przy okazji zobaczysz, że każdy z elementów tablicy został sprawdzony. Świadczy o tym console.log wewnątrz funkcji.

Innymi słowy - sprawdzana jest każda wartość w tablicy z osobna. Jeśli dana wartość jest inna niż 'Ola', zwracam ją w tablicy wynikowej.

**Podsumowanie rozdziału**

Nauczyliśmy się używać funkcji wbudowanych tablic. Ta umiejętność z pewnością bardzo Ci się przyda.

Teraz już wiesz czym są funkcje wbudowane tablic. Ich użycie jest wręcz bajecznie proste oraz zalecane.

**Zadanie: Łączymy tablice!**

Zadeklaruj dwie tablice: jedną zawierające imiona damskie oraz drugą męskie.

Połącz dwie tablice w jedną zawierającą wszystkie imiona.

Za pomocą dostępnych funkcji i przy użyciu instrukcji warunkowej spróbuj dodać jakieś imię.

**Uwaga:** jeśli imię znajduje się już w tablicy, nie może być dodane ponownie.

Sprawdź, czy wszystko działa prawidłowo. Czy dodawanie nowych imion do tablicy jest możliwe, a jednocześnie zostało zablokowane dodawanie duplikatów?

W pliku 'scripts.js' zadeklaruj dwie tablice:

var femaleNames = ['Asia', 'Kasia', 'Ola', 'Jola'];

var maleNames = ['Piotrek', 'Marek', 'Arek', 'Jarek'];

Użyj funkcji concat(), aby połączyć ze sobą powyższe dwie tablice i przypisz je do zmiennej allNames.

Poniżej zadeklaruj nową zmienną zawierającą napis var newName = 'Marian'

Stwórz instrukcję warunkową if i sprawdź, czy wartość wyrażenia allNames.indexOf(newName) === -1.

Jeśli powyższe wyrażenie jest spełnione, za pomocą funkcji push() dodaj tę zmienną do tablicy allNames.

Stwórz nowe repozytorium i wyślij tam swoje zadanie, a link do niego wklej poniżej. Pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/108236)

Początek formularza

 Wyslij link

Dół formularza

**9.3. Łańcuchy znaków**

Stringi, czyli ciągi znaków lub inaczej łańcuchy znaków, zdążyliśmy już poznać i wiesz, że można je do siebie dodawać. Warto jednak wiedzieć, że - podobnie jak obiekty tablic - typ string posiada swoje wbudowane metody i właściwości, które z pewnością Ci się przydadzą.

Deklaracja nowego stringa może odbyć się na dwa sposoby: za pomocą literału lub za pomocą new, czyli utworzenia nowego obiektu typu string.

var str = "test";

var newStr = (new String("test"));

var result1 = str.toUpperCase();

var result2 = newStr.toUpperCase();

Stringi są tak naprawdę obiektami i można je również interpretować jako zbiór znaków.

Poniżej przedstawiona jest zmienna typu string o wartości 'test' tak, jak widzi ją JavaScript.

{

0:"t",

1:"e",

2:"s",

3:"t",

length:4

}

Dzięki temu możliwe jest odwołanie się do poszczególnych jej liter na przykład za pomocą atrybutu. Choć tutaj przypomina to nieco tablice.

console.log( newStr[3] );

**length**

Jest to właściwość zwracająca długość napisu, włączając w to białe znaki, takie jak spacje.

var txt = 'Lorem ipsum dolor sit amet';

console.log(txt.length); // It will display 26

Wyświetlona wartość to ilość znaków, którą zawiera tekst. W tym wypadku jest to 26.

**charAt()**

Funkcja zwraca znak, który znajduje się na określonej pozycji w ciągu znaków (stringu). Aby sprawdzić, jaka litera znajduje się na danym miejscu, należy wskazać jej indeks jako parametr.

Trzeba pamiętać, że numerowanie zaczyna się tutaj od zera. Zupełnie jak w tablicach.

var txt ='Example';

console.log( txt.charAt(4) ); //It will display 'p'

Jeśli wskazany indeks nie będzie wskazywał na jakiś znak, to zostanie zwrócony pusty napis.

Pusty napis to string o zerowej długości.

**indexOf()**

Ta metoda zwraca położenie określonej litery lub ciągu znaków (podobnie jak w tablicach). Zwracana wartość to pierwsze napotkane wystąpienie danego ciągu.

var txt = 'Lorem ipsum dolor sit amet';

var indexDolor = txt.indexOf('dolor');

if (indexDolor == -1) {

console.log('Dolor does not appear in the txt');

} else {

console.log(txt[indexDolor]);

}

Wartość 'indexDolor' wskazuje na początek znalezionego ciągu znaków. W tym przypadku jest to 12, bo - tak jak zostało wspomniane na początku - napis jest obiektem i do poszczególnych liter można się odwołać przez indeks.

Jeśli szukany ciąg znaków w tekście nie wystąpi, zwrócona zostanie wartość '-1'. Oznacza to tylko tyle, że szukany ciąg nie występuje w zmiennej.

**slice()**

Ta metoda zwraca kawałek tekstu z napisu. Jako parametr przyjmuje indeks początkowy oraz końcowy znaków napisu. Wynikiem jest zakres znaków napisu wskazany przez parametry.

var motto = 'Good friends are loyal people';

var partOfMotto = motto.slice(5,11);

console.log(partOfMotto);

Wartością 'partOfMotto' będą znaki (jak w tablicach pierwszy znak ma indeks '0') od 5 włącznie, do 11, ale bez znaku o indeksie 11.

[5] f

[6] r

[7] i

[8] e

[9] n

[10] d

Można tutaj podać indeksy ujemne. Wtedy będzie oznaczać to liczenie indeksu od końca ciągu - tak jakby od prawej.

var motto = 'Good friends are loyal people';

partOfMotto = motto.slice(-12,-7);

console.log(partOfMotto);

Wartością 'partOfMotto' będzie napis 'loyal'. Policz indeksy od prawej.

**substr()**

Jest to funkcja podobna do metody slice, z tym, że drugi parametr zamiast indeksu ostatniego znaku przyjmuje liczbę następujących znaków napisu.

var motto = 'The rest is silence';

var mottoCharsAfter = motto.substr(12, 7);

console.log(mottoCharsAfter);

Zwrócona wartość to 7 znaków, poczynając od indeksu 12, zatem logowaną wartością jest 'silence'.

Tutaj można również podać ujemny indeks początkowy. Zasada jest taka sama, jak w funkcji 'slice'.

**replace()**

Metoda replace służy do zastępowania ciągu znaków innym ciągiem znaków. Przyjmuje zatem dwa parametry - co zamienić i na co zamienić.

var motto = 'Never give up on what you really want to do';

var mottoCharsAfter = motto.replace('do', 'have');

console.log(mottoCharsAfter);

Tutaj również obowiązuje zasada, że zamienione zostanie tylko pierwsze wystąpienie napisu.

**toUpperCase()**

Metoda nie przyjmuje parametrów - po prostu zamienia małe litery na wielkie.

var name = 'John Doe';

var nameUpperCased = name.toUpperCase();

console.log(nameUpperCased);

Wynikiem będzie 'JOHN DOE'.

**toLowerCase()**

Działanie funkcji jest odwrotne do metody 'toUpperCase'. Litery napisu zostaną zamienione na małe.

Zastosowanie powyższych dwóch metod jest szczególnie przydatne przy sprawdzaniu wystąpień jakichś słów w tekście.

var text = 'BeTTer lATe THAN neveR';

if ( text.toLowerCase().indexOf('late') != -1 ) {

console.log('Sentence contains word - late')

}

Ze względu na wielkość znaków, *lATe* jest czymś innym niż *late*. Dzięki zastosowaniu funkcji lowercase możemy sprawdzić, czy w tekście znajduje się dane słowo. W tym celu dodatkowo wykorzystujemy funkcję 'indexOf', która była omówiona wcześniej.

Wartość '-1' zostanie przez nią zwrócona tylko w przypadku, kiedy słowa 'late' nie będzie w tekście. Każda inna wartość będzie wskazywać na miejsce, w którym szukany wyraz się zaczyna.

**split()**

Metoda służy do zamiany ciągu znaków na tablicę - należy tutaj podać znak, według którego podział nastąpi. Nie musi być to podział na pojedyncze litery, ale na przykład na słowa.

Wyobraź sobie, że posiadasz listę słów kluczowych i z jakiegoś powodu chciałbyś zamiast napisu mieć tablicę słów kluczowych.

Powodem może być na przykład chęć wyświetlenia tych słów jako tagi. Za pomocą napisu ciężko byłoby to zrobić.

Długość słów jest różna, więc nie da się tak dopasować wzorca, aby to zrobić za pomocą którejś z uprzednio poznanych metod.

Zamiana ciągu znaków na tablicę, a następnie jej wyświetlenie, to w tym wypadku odpowiednie kroki. A więc do dzieła.

var keywords = 'red, green, blue, yellow, pink, orange, white';

var keywordsArray = keywords.split(', ');

Najpierw deklarujemy zmienną 'keywords', która zawiera słowa kluczowe oddzielone za pomocą przecinka oraz spacji po przecinku.

Zmienna 'keywords' tutaj jest po prostu napisem. Kluczowe jest tutaj wskazanie w jej parametrze znaku/znaków, według którego/których odbędzie się podział.

Wskazaliśmy, że podział odbędzie się według przecinka i spacji, a więc wszystko, co znajduje się pomiędzy tymi znakami, trafia do tablicy wynikowej, a ta z kolei zostaje przypisana do zmiennej 'keywordsArray'.

Skoro mamy tablicę słów, to wystarczy już tylko użyć poznanej wcześniej metody map, aby wyświetlić słowa jako tagi.

keywordsArray.map( function(element) {

console.log('#tag-' + element);

} );

**Podsumowanie rozdziału**

Wbudowane funkcje dla ciągów są po części podobne do tych dla tablic. W tym rozdziale dowiedzieliśmy się, co można robić z napisami poza ich łączeniem, a także jak sprytnie je modyfikować oraz konwertować do postaci tablic.

**Zadanie: Velociraptor czy Triceratops**

1. Poniższy tekst przypisz do zmiennej text:

Velociraptor is a genus of herbivorous ceratopsid dinosaur that first appeared during the late Maastrichtian stage of the late Cretaceous period.

1. To nie jest prawda. Powyższe zdanie dotyczy Triceratopsa :)
2. Zadeklaruj nową zmienną o nazwie `dinosaur` zawierającą tekst 'triceratops' , a następnie spraw, by stał się on 'UPPERCASE', czyli składał się z wielkich liter.
3. Następnie zamień 'Velociraptor' w tekście ze zmiennej 'text' na wartość zmiennej 'dinosaur', używając wbudowanych funkcji napisów.
4. Na koniec wyświetl za pomocą console.log() tylko pierwszą połowę otrzymanego zdania (length/2), również z użyciem funkcji napisów.
5. Spodziewany wynik to:

TRICERATOPS is a genus of herbivorous ceratopsid dinosaur that first app

1. Stwórz nowe repozytorium i wyślij tam swoje zadanie, a link do niego wklej poniżej. Pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/108237)

Początek formularza

 Wyslij link

Dół formularza

**9.4. Pętle**

Pętle wykorzystujemy, jeżeli chcemy wykonać jakiś kawałek kodu wiele razy. Najczęściej wykorzystuje się je przy pracy z tablicami lub obiektami, ale nie tylko. Wszystko zależy od celu, jaki zamierzamy osiągnąć.

Każda z pętli posiada blok instrukcji. W tym miejscu definiuje się kod, który będzie wykonywany w ramach pętli.

Pętle mogą wykonywać się określoną lub nieskończoną ilość razy. Dlatego w ich przypadku bardzo ważnym elementem jest **warunek**, który decyduje o tym ile razy funkcja się wywoła.

Czasem możesz spotkać się również z określeniem **iteracja**. Jest to określenie powtarzalności pętli. Jedna iteracja to jedno przejście pętli.

**Pętla for**

Ta pętla jest chyba najczęściej spotykana. Wszystkie parametry definiujące jej działanie, znajdują się na początku i są dobrze widoczne.

Działanie pętli opiera się o sprawdzanie wartości zmiennej po każdym przebiegu pętli (iteracji). Nazwiemy ją 'zmienna\_pętli', ponieważ ta nazwa mówi wprost, w jakim celu tworzymy tę zmienną.

Aby użyć tej pętli, należy najpierw zapisać słowo kluczowe for. Dzięki temu wiemy, z jakim rodzajem pętli mamy do czynienia.

Kolejnym elementem jest nawias okrągły, w którym zawarte są parametry pętli.

Ostatnim elementem jest zdefiniowanie bloku pętli i umieszczenie w nim kodu, który będzie wykonywany w każdym jej przebiegu.

Poniżej został zamieszczony zapis pętli z wyszczególnieniem parametrów wraz z polskim tłumaczeniem:

for ( initial\_value ; condition ; value\_modification) {

//code included in the loop block

}

for ( wartość\_inicjująca ; warunek ; modyfikacja\_wartości) {

//kod zawarty w bloku pętli

}

**Opis parametrów pętli for:**

* **wartość inicjująca** - deklaracja 'zmiennej\_pętli', która będzie definiować działanie pętli oraz nadanie jej początkowej wartości,
* **warunek** - operacja logiczna (zwraca true albo false), która definiuje, czy pętla ma się dalej wykonywać, czy zostać przerwana,
* **modyfikacja\_wartości** - z uwagi na to, że pętla sterowana jest wartością 'zmiennej\_pętli', ostatnim parametrem jest operacja modyfikująca jej wartość. Najczęściej spotykana jest tutaj inkrementacja bądź dekrementacja.

A dokładniej, krok po kroku:

1. Zainicjowanie wartości początkowej.
2. Sprawdzenie warunku.
3. Jeśli warunek jest true, wykonaj krok 4., a jeśli false, wykonaj krok 6.
4. Wykonaj blok pętli
5. Zmodyfikuj wartość 'zmiennej\_pętli' i wykonaj krok 2.
6. Przestań wykonywać pętlę i wróć do skryptu.

Ten rodzaj pętli może nie wykonać się ani razu, bo warunek jest sprawdzany na samym początku, a następnie po każdym jej przejściu. Może również działać w nieskończoność, jeśli warunek zostanie niewłaściwie zdefiniowany.

Przykład:

for ( var k = 0 ; k < 10 ; k++) {

console.log('The value of k is now: ' + k);

}

Powyższa pętla wyświetla wartość bieżącą 'zmiennej\_pętli'. W tym przypadku zostaną wyświetlone wartości od 0 do 9.

**Co się tutaj dzieje?**

Po pierwsze, zostaje zainicjowana 'wartość\_inicjująca', czyli zmienna k, która na początku przyjmuje wartość '0' (var k = 0).

Następnie jest sprawdzany 'warunek' - 'czy k jest mniejsze od 10 (k < 10)'. Na razie tak, więc wykonany jest kod pętli. Jeśli warunek byłby niespełniony, pętla nie wykonałaby się ani razu.

Kolejnym krokiem jest inkrementacja. Wartość k zostaje zwiększona o 1 ('k++'), a następnie sprawdzany jest warunek.

Czy 'k' jest mniejsze od 10?

Dopóki odpowiedzą będzie tak, kod pętli będzie wykonywany.

I tak w kółko, aż do momentu, kiedy wartość 'k' wyniesie 10 - wtedy warunek nie zostanie spełniony, a pętla zostanie opuszczona.

Przeanalizuj powyższy przykład, biorąc pod uwagę przedstawioną szczegółową instrukcję.

**Pętla while**

While ma o wiele prostszą budowę niż pokazana poprzednio pętla for, choć sama budowa pętli jest podobna. Jednak w tym przypadku jedynym parametrem jest 'warunek'. Aby zapisać pętlę - podobnie jak poprzednio - należy zapisać słowo kluczowe while, następnie w nawiasach okrągłych 'warunek' oraz blok pętli.

Poniżej widnieje zapis pętli z wyszczególnieniem parametrów i kluczowych elementów:

while (condition) {

...

}

Warunek powinien być operacją logiczną, która zwróci true lub false. Z uwagi na to, że w samej pętli nie definiuje się dodatkowych zmiennych, warunek będzie korzystał z innych dostępnych zmiennych.

Po wykonaniu kodu pętli, należy pamiętać o modyfikacji zmiennej, która będzie wykorzystana w 'warunku'.

**Przykładowa pętla:**

var i = 0;

while (i < 10) {

console.log(i);

i++;

}

Ten rodzaj pętli od razu sprawdza warunek. Z tego względu może nie wykonać się ani razu.

**Krok po kroku:**

1. Sprawdź warunek pętli. Jeśli jest true, przejdź do kroku 2., w przeciwnym wypadku przejdź do kroku 3.
2. Wykonaj kod pętli i wróć do kroku 1.
3. Zakończ działanie pętli.

**Co się tutaj dzieje?**

Deklarujemy zmienną i oraz nadajemy jej początkową wartość.

Następnie natrafiamy na pętlę while, w której sprawdzany jest 'warunek' pętli. Jeśli nie zostanie spełniony (false), pętla zostanie pominięta. W przypadku jego spełnienia (true), pętla zostaje wykonana, a następnie ponownie jest sprawdzany warunek.

Dzieje się tak aż do momentu, kiedy i osiąga wartość 10. Wtedy warunek stanie się niespełniony (false), a pętla przerwie swoje działanie.

Gdyby nie inkrementacja wewnątrz bloku, wartość i nie zmieniała by się, przez co zawsze sprawdzany byłby warunek "Czy 0 jest mniejsze od 10", a pętla wykonywałaby się w nieskończoność. Albo do momentu, gdy zabrakłoby prądu.

**Pętla do while**

Ten rodzaj pętli najpierw się wykonuje, a dopiero później sprawdza warunek. Sprawia to, że pętla zostanie wykonana co najmniej jeden raz, nawet jeśli warunek od początku byłby niespełniony.

Aby zdefiniować pętlę do while, należy najpierw zapisać słowo kluczowe "do", następnie blok pętli i słowo kluczowe "while", po którym zdefiniowany jest w nawiasach okrągłych warunek.

Poniżej znajduje się zapis pętli z wyszczególnieniem jej kluczowych elementów:

do {

//loop code

} while (condition)

Warunek, jak dotychczas, musi dawać wartość true lub false.

**Krok po kroku:**

1. Wykonaj kod pętli.
2. Sprawdź warunek. Jeśli jest on prawdziwy, wykonaj krok 1., a jeśli nie, to krok 3.
3. Zakończ działanie pętli.

Podobnie jak w przypadku pętli while, trzeba zadbać o to, aby zmienna wykorzystana w warunku zmieniała swoją wartość w przebiegu pętli.

**Przykład:**

var i = 0;

do {

console.log(i);

i++;

} while (i < 10);

**Co się tutaj dzieje?**

Zostaje zadeklarowana zmienna i o wartości 0. Następnie wykonywany jest kod pętli, w którym najpierw wyświetlona zostaje początkowa wartość zmiennej, a następnie wartość zmiennej i zwiększona o 1.

Ostatnim krokiem jest sprawdzenie warunku. Pętla jest wykonywana, dopóki warunek jest prawdziwy.

**Pętla for in**

To specjalny rodzaj pętli dzięki któremu możemy poruszać się po atrybutach obiektu.

Aby ją zadeklarować, należy użyć słowa kluczowego "for". Następnie w nawiasach okrągłych trzeba zdefiniować 'zmienną\_pętli', ale bez nadawania jej wartości. Wreszcie musimy wskazać obiekt, na którym ma wykonać się pętla, a także blok pętli, w którym zostanie zawarty kod mający się powtarzać w ramach pętli.

Poniżej widnieje zapis pętli z wyszczególnieniem parametrów i kluczowych elementów wraz z tłumaczeniem:

for (loop\_variable in indicated\_object) {

// code in the loop block

}

for (zmienna\_pętli in wskazany\_obiekt) {

// kod w bloku pętli

}

Pętla for in posiada dwa parametry. Pierwszym z nich jest zmienna pętli, służy ona do przechowania bieżącej wartości pętli. Drugim, po słowie kluczowym "in" jest obiekt, który zamierzamy iterować po atrybutach.

Kod pętli będzie wykonany tyle razy, ile atrybutów ma wskazany obiekt.

**Przykładowy obiekt:**

var person = {

name: 'Piotr',

age: 25,

country: 'Poland'

};

**Przykład pętli:**

for (var attrName in person) {

console.log(attrName);

}

Pętla, wraz z każdym swoim przejściem, czyta kolejny atrybut obiektu i zapamiętuje jego nazwę w 'zmiennej pętli'.

W tym przypadku pętla wykona się więc 3 razy.

Zmienna 'attrName' kolejno przyjmie wartości 'name', następnie 'age' oraz 'country'.

W jakim celu istnieje pętla for in?

W tablicach było wiadomo, że indeksami będą cyfry. W obiektach może być to dowolny ciąg znaków, a w praktyce wygląda to tak, że nigdy nie wiadomo, jakie atrybuty posiada obiekt i do jakich nazw można się odwołać.

Ten rodzaj pętli pozwala dosłownie przeglądać obiekt.

object[attrName];

**Instrukcja przerwania**

Ta operacja nie jest funkcją, jest to specjalne polecenie, które ma zastosowanie w instrukcji switch oraz dowolnej pętli.

Instrukcja przerwania to polecenie break;.

for ( var k = 0 ; k < 10 ; k++) {

console.log('The value of k is now: ' + k);

if (k == 5) {

console.log('k is equal to 5. The end');

break;

}

}

Pętla for w powyższym przykładzie zgodnie z zapisem dąży do wykonania się 10 razy (0 - 9).

Jednak w jej wnętrzu umieszczona jest instrukcja warunkowa, która wykona się w momencie kiedy 'k' będzie miało wartość '5'.

W tym momencie zostanie wykonane polecenie break; a pętla zostanie przerwana.

**Instrukcja przeskoku**

Ta instrukcja, podobnie jak break, również nie jest funkcją - jest to specjalne polecenie, które ma zastosowanie w instrukcji switch oraz dowolnej pętli.

Instrukcja przeskoku to polecenie continue;.

for ( var k = 0 ; k < 10 ; k++) {

console.log('The value of k is now: ' + k);

if (k > 5) {

continue;

}

console.log('Instruction at the end of the loop ');

}

Polecenie continue; nie przerywa pętli, a jedynie pomija tę część kodu w bloku pętli, która znajduje się za tym poleceniem.

Sama pętla nie zostaje przerwana.

W powyższym przykładzie pętla powinna wyświetlić dwa komunikaty, w każdej iteracji. Jednak kiedy wartość k, będzie większa niż 5, zostanie wykonane polecenie 'continue', które sprawi, że kod znajdujący się poniżej 'continue' zostanie pominięty.

**Podsumowanie rozdziału**

Wiesz już, czym są pętle i jak ich używać. Potrafisz wykonać pewne operacje określoną ilość razy lub aż do określonego warunku przerwania pętli.

Potrafisz przejrzeć atrybuty dowolnego obiektu. Wiesz, jak warunkowo przerwać wykonywanie kodu pętli bez jej kończenia lub opuścić ją w dowolnym momencie bez względu na podany warunek.

To kolejne z możliwości, które mogą Ci się przydać w najróżniejszych sytuacjach.

Kiedy rozwiązań jest wiele, musisz pamiętać, że każde z nich może być dobre i to Twoim zadaniem jest wybrać najodpowiedniejsze.

**Zadanie: Choinka**

Napisz funkcję, która będzie wyświetlać choinkę dowolnej wysokości w konsoli.

**Polecenie:**

drawTree(1)

ma dać wynik:

\*

**Polecenie:**

drawTree(2)

ma dać wynik:

\*

\*\*

**Polecenie:**

drawTree(5)

ma dać wynik:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

1. Zadeklaruj funkcję drawTree, która przyjmuje jeden parametr, ten parametr będzie odzwierciedlał wysokość choinki (liczbę wypisanych wierszy w konsoli).
2. Wewnątrz funkcji zadeklaruj pętlę for, która wykona się tyle razy, ile mamy poziomów (parametr przekazany do drawTree odpowiada za liczbę rysowanych wierszy). Żeby zachować porządek, zmienną której będziemy używać do działania pętli nazwij i
3. Wewnątrz tej pętli, utwórz zmienną star, która będzie pustym stringiem (stan początkowy choinki, który będzie rozbudowywany w kolejnej pętli).
4. Następnie, wciąż wewnątrz głównej (zewnętrznej) pętli, zadeklaruj kolejną pętlę for. Tak, pętla w pętli! Tym razem zmienną, którą będziemy liczyć iteracje, nazwij j
5. Wewnętrzna pętla ma się wykonać zależnie od tego, który wiersz aktualnie drukujemy w konsoli. Czyli od tego, która to iteracja głównej pętli. Warunek stopu to aktualny poziom (wartość zmiennej i).
6. Wewnątrz drugiej pętli nadpisz zmienną star i za każdym razem kiedy pętla będzie wykonana, do aktualnej wartości star dodaj kolejną gwiazdkę star += '\*';.
7. Po opuszczeniu wewnętrznej pętli wyświetl console.log(star).
8. Stwórz nowe repozytorium i wyślij tam swoje zadanie, a link do niego wklej poniżej. Pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

Dla ambitnych: na każdym z poziomów dodaj spacje tak, aby choinka stała prosto. Przed rozpoczęciem pisania tej części zadania zastanówmy się, jak nasza choinka będzie wyglądała.



Pętla główna for, która wyświetla nam kolejne poziomy, na pewno zostaje. Wewnątrz niej moglibyśmy umieścić dwie kolejne: jedną rysującą spacje, a drugą rysującą gwiazdki. Teraz wystarczy zastanowić się nad kolejnymi poziomami choinki.

* POZIOM 1: \_ \_ \*
* POZIOM 2: \_ \* \* \*
* POZIOM 3: \* \* \* \* \*

Wraz ze wzrostem poziomu: pętla rysująca spacje zmniejsza się o jeden, a pętla rysująca gwiazdki rysuje ich o dwie więcej. Teraz trzeba uzależnić obie pętle wewnętrzne od wysokości choinki. Jeśli dalej masz problem z zadaniem dla ambitnych, wyciągnij kartkę papieru i rozpisz na niej jak zachowują, albo powinny zachowywać się poszczególne zmienne użyte w kodzie. Powodzenia!

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/108238)

Początek formularza

 Wyslij link

Dół formularza

**9.5. DOM**

**DOM** (czyli Obiektowy Model Dokumentu, ang. *Document Object Model*) to API umożliwiające przetwarzanie, modyfikowanie, uaktualnianie i usuwanie części dokumentu HTML.

API (Interfejs Programowania Aplikacji) pozwala z kolei na korzystanie z funkcjonalności udostępnianych przez DOM w twojej aplikacji. Jest to pewien zbiór funkcji oraz właściwości, które daje Ci przeglądarka.

Po wczytaniu dokumentu HTML przeglądarka przeprowadza następujące czynności:

* analizuje składniowo kod HTML,
* ładuje zewnętrzne skrypty i arkusze stylów,
* tworzy drzewo DOM,
* ładuje obrazki i pozostałe zewnętrzne elementy.

Każda przeglądarka implementuje DOM API, dlatego zawsze mamy dostęp do drzewa dokumentu HTML.

Poniżej przedstawione są najważniejsze właściwości udostępniane przez DOM.

**window**

Obiekt *window* tworzony jest automatycznie w chwili otwarcia okna przeglądarki, a wskazuje na bieżące okno bądź zakładkę.

Posiada on szereg właściwości (wartości dotyczących okna przeglądarki) i metod (funkcji) przydatnych podczas tworzenia strony.

Ponieważ obiekt *window* jest najważniejszym obiektem w hierarchii, to w odwołaniu do jego właściwości i metod nie trzeba dodawać nazwy obiektu, chyba że ma to być odwołanie do innego okna (nie bieżącego).

Obiekt *window* posiada takie metody jak: setTimeout(), setInterval, localStorage, history, a także właściwości/obiekty scrollTop, outerHeight, outerWidth - jest tego naprawdę sporo.

Jeśli ciekawi Cię, co dokładnie posiada ten obiekt, zapraszam Cię do wykonania poniższego kodu.

console.log(window);

Tak, to takie proste ;)

**setTimeout**

Ta metoda przyjmuje dwa parametry. Pierwszym z nich jest funkcja, która ma zostać wykonana z opóźnieniem, a drugim czas opóźnienia wykonania tej funkcji (w milisekundach).

Wywołanie tej funkcji sprawia, że wykonanie funkcji będącej pierwszym parametrem odbędzie się po określonym czasie.

Na przykład jeśli chcesz wyświetlić coś po 2 sekundach od załadowania strony, to należy wpisać poniższy kod:

function raiseAlarm() {

alert('I show it once after 2 seconds.');

}

var delayedAlert = setTimeout(raiseAlarm, 2000);

**clearTimeout**

Funkcja ta przerywa wykonanie funkcji, która została zaplanowana do wykonania z opóźnieniem.

Jako parametr przyjmuje ona wskazanie na wywołanie funkcji setTimeout(). Aby mieć to wskazanie, najlepiej jest przypisać wywołanie funkcji setTimeout-a do zmiennej, jak w poniższym przykładzie:

var delayedAlert = setTimeout(raiseAlarm, 2000);

clearTimeout(delayedAlert);

Funkcja, której wywołanie zostało opóźnione, nie zostanie wcale wykonana, ponieważ za pomocą funkcji clearTimeout() anulujemy chęć jej wykonania.

**clearInterval**

Funkcja clearInterval() odnosi się do funkcji setInterval (która zostanie zaraz opisana) w taki sam sposób, jak setTimeout, z tym że sprawia, że określona funkcja będzie wykonywana wielokrotnie, co zdefiniowany odstęp czasu.

Wykonanie clearInterval, podobnie jak clearTimeout, wymaga wskazania na to wywołanie funkcji i po prostu anuluje chęć jej wykonywania w odstępach czasu.

Zanim pokażemy Ci nieco ciekawszy przykład zastosowania tych funkcji, musisz wiedzieć, że taka funkcja istnieje oraz że przerywa działanie funkcji setInterval. Przykład znajduje się poniżej:

**setInterval**

Metoda działa tak samo jak setTimeout, z tym że wykonuje się wielokrotnie w określonym odstępie czasu, czyli na przykład raz na dwie sekundy.

function raiseAlarm() {

alert('I show myself every two seconds.')

}

var interval = setInterval(raiseAlarm, 2000);

function stop() {

alert('It took 10 seconds. The following statement interrupts the interval.');

clearInterval(interval);

}

setTimeout(stop, 10000);

Na początku definiujemy funkcję raiseAlarm, która ma za zadanie wyświetlić tekst. Chcemy, aby ta funkcja wykonywała się co dwie sekundy, dlatego wywołujemy funkcję setInterval(raiseAlarm, 2000). Pierwszym jej parametrem jest wskazanie na funkcję raiseAlarm, natomiast drugim jest odstęp czasu w milisekundach SetInterval sprawi, że funkcja raiseAlarm będzie wykonywana co dwie sekundy(2000ms).

Interwał również zostaje zapamiętany w zmiennej interval, tak aby można go było przerwać w przyszłości.

Kolejno zdefiniowana zostanie funkcja stop, która wyświetli komunikat i przerwie uprzednio wystartowany interwał. Ale stanie się tak dopiero, kiedy funkcja stop zostanie wywołana.

Na koniec nastawiamy setTimeout, aby wykonał funkcję stop za kolejne 10 sekund i przerwał nasz interwał za pomocą metody clearInterval.

W ten sposób sprawiliśmy, że funkcja raiseAlarm była wykonywana regularnie przez określony czas.

**document**

Obiekt *document*, a właściwie *window.document*, należy do obiektu *window*, jednak ze względu na swoje możliwości zasługuje na wyszczególnienie.

Posiada on szereg właściwości i metod przydatnych podczas tworzenia strony - jedne z najważniejszych opiszę poniżej.

Zanim zaczniemy testować co tutaj można zrobić, stwórzmy sobie dokument HTML z następującymi tagami.

<div class="content">

<div id="nav">

<ul>

<li class="menuItem">Homepage</li>

<li class="menuItem">About me</li>

<li>Contact</li>

</ul>

</div>

<div id="footer">

<ul>

<li class="menuItem">SocialLink</li>

</ul>

</div>

</div>

Pamiętaj, aby dołączyć skrypt do tej strony. Nie piszę jak, bo to już z pewnością wiesz - jeśli nie pamiętasz, zapraszam do sekcji "Pierwsze kroki" lub "Początki są zawsze trudne".

console.log(document); // or window.document

Obiekt ten przechowuje całą strukturę strony, udostępnia także szereg metod i właściwości służących manipulacji elementami strony, o czym jeszcze zdążysz się przekonać.

**onload**

Ta metoda zostaje wykonana tuż po załadowaniu się strony. Aby mieć pewność, że nie odwołamy się do elementu na stronie, który nie zdążył powstać, kod sktyptu należy umieścić wewnątrz poniższej funkcji.

window.onload = function() {

console.log('The page is fully loaded.');

// the rest of the script

}

**location**

Obiekt location odnosi się do bieżącego adresu URL.

console.log(location);

Gdybyś chciał kiedyś przekierować użytkownika na inną stronę z użyciem JavaScriptu, to będzie to możliwe właśnie dzięki temu obiektowi.

window.location = 'http://www.kodilla.pl';

Sprawdź, jak to działa.

Obiekt *location* udostępnia również m.in. metody/właściwości służące do wymuszenia przeładowania strony i pobrania domeny strony.

**getElementById**

Metoda, która pobiera element o id pobranym w argumencie.

Jeśli mamy więcej niż jeden element z takim samym id, zwrócony zostanie tylko pierwszy znaleziony.

var navigation = document.getElementById('nav');

console.log(navigation);

Powyższy przykład wymaga, aby na stronie istniał element z przekazanym id - przykład poniżej:

<div id="nav">Navigation element</div>

**getElementsByClassName**

Pobranie elementu z dokumentu po nazwie klasy - ważna sprawa. Tutaj, niezależnie od ilości elementów, dostajemy tablicę elementów - to znaczy, że do poszczególnych elementów należy odwołać się przez indeks w tablicy.

Uwaga - mamy tutaj słowo **Elements**, a przy pobieraniu po id mamy **Element** - to częsta przyczyna błędów w kodzie i warto o tym pamiętać.

var withMenuItemsClass = document.getElementsByClassName('menuItem');

console.log(withMenuItemsClass);

Do zbioru trafiły wszystkie elementy posiadające klasę menuItem niezależnie od położenia na stronie.

<ul id="Menu">

<li class="menuItem">item 1</li>

<li class="menuItem">item 2</li>

<li class="menuItem">item 3</li>

</ul>

Przeszukiwać można także w poszczególnych elementach DOM-u - każdy z elementów udostępnia te same metody.

var menu = document.getElementById('Menu');

var withMenuItemsClassInNav = menu.getElementsByClassName('menuItem');

console.log(withMenuItemsClassInNav);

W tym przypadku otrzymaliśmy również elementy posiadające klasę menuItem, ale tylko te znajdujące się w elemencie menu, który został wyszukany na początku.

**getElementsByTagName**

Ta metoda działa analogicznie do getElementsByClassName, z tym że szukamy po określonych tagach.

var itemsByTagName = document.getElementsByTagName('li');

console.log(itemsByTagName);

**nextElementSibling**

Ta metoda pomaga poruszać się po elementach. Wskazuje ona na kolejny element znajdujący się na tym samym poziomie zagnieżdżenia (w CSS używaliśmy do tego znaku `+` w selektorze).

var navigation = document.getElementById('nav');

console.log(navigation.nextElementSibling);

<div id="wrapper">

<div id="nav">

</div>

<div id="footer">

</div>

</div>

W tym przypadku zostanie wskazany element z id footer.

**createElement**

Umożliwia stworzenie nowego elementu. Jako parametr należy przekazać nazwę tagu bez nawiasów.

var newElem = document.createElement('p');

console.log(newElem);

Ten element został utworzony i znajduje się w pamięci, jednak na razie nie został dołączony do dokumentu.

**innerHTML**

Ta właściwość umożliwia wstawienie kodu HTML do wnętrza wskazanego elementu. Posłużymy się poprzednim przykładem.

var newElem = document.createElement('p');

newElem.innerHTML = 'The text inside the p tag, which is under newElem';

console.log(newElem);

Ten element wciąż znajduje się w pamięci, jednak na razie nie został dołączony do dokumentu.

Tak jak poniżej:

<p>The text inside the p tag, which is under newElem</p> 

**appendChild**

Dzięki tej metodzie będzie możliwe dołączenie dowolnego elementu na końcu elementu, na którym wywołana jest metoda. Być może wydaje się to nieco skomplikowane, dlatego omówimy to na przykładzie.

Mamy dany poniższy kod HTML:

<div id="nav"></div>

Następnie zostaje wykonany skrypt:

var navigation = document.getElementById('nav');

var newElem = document.createElement('p');

newElem.innerHTML = 'The text inside the p tag, which is under newElem';

navigation.appendChild(newElem);

Najpierw wskazujemy na element z identyfikatorem nav (to ten, który już istnieje w strukturze HTML) i przypisujemy go do zmiennej navigation.

Następnie utworzony zostaje nowy, pusty element <p></p>, który przypisujemy do zmiennej newElem. Dzięki tej zmiennej możemy się odwołać do nowo utworzonego elementu.

Za pomocą właściwości innerHTML wewnątrz newElem umieszczony zostaje tekst.

Na tym etapie w pamięci pod adresem zmiennej newElem istnieje element

<p>Tekst wewnątrz tagu<p>, który znajduje się pod zmienną newElem

Ostatnim krokiem jest wywołanie funkcji appendChild i przekazanie jej elementu. appendChild sprawia, że przekazany element zostaje dołączony na koniec elementu, na którym został wywołany.

Z polskiego na polski - do elementu navigation zostaje dołączony nowy element potomny - newElem.

**className**

Umożliwia przeczytanie atrybutu class danego elementu.

Możliwe jest także dodanie nowych klas za pomocą tej metody.

Obowiązują tutaj takie same operacje jak na napisach.

Przykładowy kod HTML:

<div id="Menu" class="nav">

[...]

</div>

Kod JavaScript:

var menu = document.getElementById('Menu');

menu.className += ' navbar-right';

console.log(menu.className);

Każdy element DOM posiada właściwość className, nawet jeśli jest ona pusta. classNamewskazuje na wartość HTML-owego atrybutu class="klasa".

W powyższym przykładzie na początku pobieramy element posiadający klasę id Menu z dokumentu.

Następnie modyfikujemy wartość atrybutu className. Po prostu dopisujemy do niego ciąg znaków (pamiętamy o spacji, aby rozdzielić klasy).

Wynikiem powyższego kodu będzie dodanie nowej klasy do elementu Menu.

A oto wynik:

<div id="Menu" class="nav navbar-right">

[...]

</div>

**style**

Umożliwia przeczytanie atrybutu style danego elementu.

Tutaj nie znajdziesz właściwości nadanych w pliku *style.css*, a jedynie te umieszczone w tagu style w danym elemencie.

*Style* jest obiektem, który w przypadku nadanych wartości jest przerabiany na string i umieszczany w tagu.

var navigation = document.getElementsByClassName('menuItem');

console.log(navigation[0].style);

navigation[0].style.background = 'red';

navigation[0].style.padding = '10px';

console.log(navigation[0].style);

Jeśli teraz sprawdzisz, jak wygląda element, to ukaże Ci się taki widok:

<li class="menuItem" style="padding: 10px; background: red;">Strona główna</li>

**Podsumowanie rozdziału**

Wiesz, czym jest DOM oraz obiekty window oraz document. Potrafisz wywoływać funkcję z określonym odstępem czasu, a także wykonać przekierowanie na inną stronę.

Potrafisz pobierać, tworzyć, dodawać oraz modyfikować elementy strony z wykorzystaniem samego JavaScriptu. Wiesz też, jak sprawdzić, jakie metody/właściwości posiada obiekt *window*.

**Zadanie: Iterowanie po elementach DOM**

Dodaj kilka elementów button do strony. Nadaj im klasę button. Za pomocą pętli for przejdź przez wszystkie te elementy i wyświetl tekst zawarty w każdym z nich za pomocą alert().

<button class="button">Hello!</button>

W pliku index.html dodaj kilka elementów <button> na stronie, zgodnie z powyższym schematem.

Użyj metody document.getElementsByClassName aby pobrać elementy ze strony i przypisz je do zmiennej.

Stwórz pętlę for, która wykona się tyle razy, ile jest elementów na stronie - przyda ci się do tego właściwość length.

Wewnątrz pętli wykonaj instrukcję, która wyświetli tekst zawarty w każdym z tych buttonów (użyj właściwości innerText).

Stwórz nowe repozytorium i wyślij tam swoje zadanie, link do niego wklej poniżej i pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/108239)

Początek formularza

 Wyslij link

Dół formularza

**9.6. Eventy**

Eventy to zdarzenia/czynności, które użytkownik wykonuje podczas odwiedzania strony. Przykładowymi zdarzeniami mogą być np. przesunięcie kursora na jakiś element (mouseover), kliknięcie czegoś (click), wysłanie formularza (submit) naciśnięcie klawisza (keypress).

Zdarzenia należy jakoś obsłużyć - w tym celu należy dodać listener, czyli nasłuchiwacz danego zdarzenia.

Aby dodać listener, należy wykonać funkcję addEventListener na danym elemencie. Funkcja ta wymaga jednak podania typu eventu oraz funkcji (*handler*), która ma zostać wykonana, gdy dane zdarzenie wystąpi.

przyciskNaStronie.addEventListener(typ, handler);

Typów zdarzeń jest wiele. Można również definiować własne typy eventów. Na początek musimy znać poniższe:

* click - kiedy element został kliknięty,
* keypress - przycisk na klawiaturze został wciśnięty
* mouseenter - kursor myszy najechał na obszar elementu,
* mouseleave - kursor myszy opuścił obszar elementu,
* mousedown - lewy przycisk myszy został wciśnięty,
* mouseup - lewy przycisk myszy został puszczony,
* resize - szerokość lub wysokość strony (okna przeglądarki) została zmieniona,
* scroll - strona/element został przesunięty za pomocą scrolla,
* load - wczytywanie elementu/strony zostało zakończone.

Rodzajów zdarzeń jest o wiele więcej.

Zapraszam Cię do zapoznania się z nimi na tej stronie:

<http://www.w3schools.com/jsref/dom_obj_event.asp> 

Kiedy nadajemy event na elemencie DOM, to będzie on zawsze dostępny pod pierwszym parametrem handlera. Posiada on różne informacje dotyczące tego, co się wydarzyło, między innymi rodzaj eventu, element, który został kliknięty, położenie kliknięcia czy wielkość strony.

Sam kliknięty element jest dostępny pod atrybutem target.

przyciskNaStronie.addEventListener(typ, function(e) {});

Widoczny parametr e jest tutaj elementem DOM, który został kliknięty.

Eventy można nadawać na dowolne części strony lub całe okno przeglądarki (*window*).

Posłużmy się poniższą strukturą HTML:

<div class="content">

<img id="js-image" src="http://placehold.it/200x200" alt="" />

<br>

<input id="js-inputValue" type="text" />

<br>

<button onclick="checkOnclickEvent('test')" id="js-button">

Click me!

</button>

</div>

Event może również zostać nadany bezpośrednio w HTML, tak jak zostało to zrobione na elemencie <button> - spójrz na dodany atrybut onclick. Należy tutaj pamiętać o tym, aby funkcja użyta w atrybucie była stworzona.

Lepszą praktyką jest robienie tego tylko i wyłącznie w pliku JavaScriptu za pomocą metody addEventListener - tak jak na poniższym przykładzie:

function checkOnclickEvent(param) {

console.log(param);

}

var inputElem = document.getElementById('js-inputValue'),

imageElem = document.getElementById('js-image');

imageElem.addEventListener('click', function(e) {

checkOnclickEvent('image was clicked')

});

inputElem.addEventListener('keypress', function(e) {

e.target.value += ' test ';

});

Powyższy fragment kodu odwołuje się do wcześniejszej struktury HTML.

Najpierw została zadeklarowana funkcja checkOnclickEvent, która wyświetla za pomocą console.log wartość parametru, jaki otrzyma.

Następnie z dokumentu pobrane zostały elementy js-inputValue, czyli po prostu <input>oraz js-image, który jest obrazkiem <img>.

W dalszym kroku nadajemy listener na obrazek. Zdarzeniem jest click, więc za każdym razem, kiedy element zostanie kliknięty, zostanie wykonana funkcja przekazana jako drugi parametr.

Wywołuje ona metodę checkOnclickEvent przekazując przy tym tekst image was clicked. To wszystko jeśli chodzi o obrazek.

Na element input dodajemy listener typu keypress, a więc funkcja wywoła się po kliknięciu jakiegoś przycisku na klawiaturze, kiedy input będzie aktywny (aby zacząć w nim pisać, trzeba go najpierw kliknąć).

Akurat ten listener ma za zadanie po każdej literze dopisać dodatkowy ciąg znaków do elementu input.

e.target wskazuje na element input na stronie, a value jest atrybutem tego inputa, który definiuje co jest w nim wpisane.

Częstą praktyką w przypadku nadawania zdarzeń na elementach w kodzie JavaScript jest nadawanie id z przedrostkiem js-, aby łatwo można było odnaleźć element na stronie.

**Podsumowanie rozdziału**

Potrafisz przypisać zdarzenia do elementów strony na dwa sposoby - z wykorzystaniem atrybutu oraz z wykorzystaniem funkcji wywołanej na elemencie z poziomu skryptu.

**Zadanie: Manipulowanie DOM po kliknięciu**

Dodaj do strony listę elementów (<ul>) oraz przycisk. Kliknięcie w niego będzie powodowało dodawanie elementów do listy.

<ul id="list">

<li>item 0</li>

</ul>

<button id="addElem">

Dodaj

</button>

Spraw, aby po kliknięciu buttona do listy został dodany elementy z napisem item oraz kolejnym numerem (tj. 0, 1, 2...)

Spodziewane elementy listy po kliknięciu buttona kilkukrotnie to:

<ul id="list">

<li>item 0</li>

<li>item 1</li>

<li>item 2</li>

</ul>

Użyj funkcji addEventListener, aby nadać listener na przycisk.

Zapamiętaj w zmiennej list odniesienie do elementu z id list.

var list = document.getElementById('list')

To samo zrób dla elementu <button> - przypisz go do zmiennej o nazwie add.

Dodaj listener do buttona - ma on reagować na kliknięcie, czyli na rodzaj eventu click.

add.addEventListener('click', function() {});

Wewnątrz funkcji, która została przekazana w listenerze, dodaj kod odpowiadający za dodawanie elementów do listy.

W tym celu wewnątrz funkcji zawartej w listenerze zdeklaruj zmienną element i za pomocą metody createElement() utwórz kolejne pozycje menu:

var element = document.createElement('li');

Teraz zajmiemy się tekstami wewnątrz utworzonych elementów li. Za pomocą element.innerHTML = 'item'; możemy przypisać tekst *item*, a następnie w odpowiednim miejscu wstawić numer bieżącego elementu.

Możesz w tym celu skorzystać z funkcji getElementsByTagName oraz właściwości length, aby dowiedzieć się, ile elementów ma lista ma w danym momencie.

Ostatnim krokiem będzie dodanie naszego elementu do zmiennej list. Do tego użyjemy funkcji appendChild(). Powinno to wyglądać tak: list.appendChild(element);. Pamiętaj żeby wszystko wykonywało się wewnątrz funkcji wykonywanej po kliku.

Stwórz nowe repozytorium i wyślij tam swoje zadanie, a link do niego wklej poniżej. Pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/108240)

Początek formularza

 Wyslij link

Dół formularza

**9.7. Wspólnie napiszmy grę w JavaScripcie**

W tym rozdziale stworzysz swoją pierwszą prostą grę. Będzie to klasyczna gra w kamień, papier i nożyce - wszyscy ją doskonale znamy. Jest ona przy tym idealnym przykładem, w którym możemy wykorzystać poznane wcześniej możliwości.

Do budowy interfejsu skorzystamy ze znanych nam rozwiązań, czyli Font Awesome oraz Twitter Bootstrap.

**Założenia gry**

Założenia są takie, jak w klasycznej grze "Kamień, papier i nożyce":

* Gra ma się składać z dowolnej liczby rund. Koniec rozgrywki następuje, gdy jeden z zawodników zdobędzie 10 punktów. Remis nie daje punktów, a wygrana daje 1 punkt dla zwycięzcy danej rundy.
* Tuż po uruchomieniu strony gracz ma zobaczyć tytuł gry oraz przycisk 'New Game'.
* Po kliknięciu przycisku 'New Game', użytkownik zostanie poproszony o podanie swojego imienia.
* Następnie gra zostanie uruchomiona, a na ekranie zamiast przycisku 'New Game' pojawią się przyciski 'Rock', 'Paper', 'Scissors' oraz tabela przedstawiająca wybór graczy (kto co wybrał), a także rezultat danej rundy.
* W tabeli mają zostać wyświetlone imię gracza oraz przeciwnika, w tym wypadku ma to być to 'Computer', oraz uzyskany wynik.
* Po zdobyciu przez któregoś gracza 10 punktów zamiast przycisków 'Rock', 'Paper', 'Scissors' ma pojawić się napis "The winner is [nazwaGracza]", a poniżej przycisk 'Play again' o takiej samej funkcjonalności jak przycisk 'New Game'.

**Mechanika gry**

Zbiór elementów, czyli tabela graczy, dostępne przyciski i wynik rezultatu, jest stały.

Wszystkie elementy, które będą wyświetlane na stronie, będą istnieć przez cały czas rozgrywki, jednak będą ukrywane lub wyświetlane zależnie od potrzeb. Dlatego HTML można zakodować od razu.

W czasie rozgrywki będziemy nanosić aktualny wynik gry oraz pobierać wybór gracza za pomocą zdarzeń oraz modyfikacji elementów .

Wybór komputera będzie losowany za pomocą wbudowanej funkcji 'random' z puli możliwych wyborów.

Zamiast dodawać dwa osobne przyciski o tej samej funkcjonalności, po prostu zmienimy wyświetlany napis w tym samym elemencie.

**Struktura HTML**

Na początek stwórz nowy branch od brancha 'master'. Pamiętaj, żeby zadanie wykonywać jako osobny projekt.

**Dołączenie zewnętrznych arkuszy stylów**

W pliku 'index.html' w sekcji '' załącz potrzebne pliki stylów Bootstrapa oraz FontAwesome.

<head>

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" rel="stylesheet" />

<link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.6.3/css/font-awesome.min.css" rel="stylesheet" />

</head>

Kolej na sekcję <body>.

**Tytuł gry**

<div class="container">

<h3 class="text-center">

Welcome to the game

<span class="label label-default">Rock</span>

<span class="label label-info">Paper</span>

<span class="label label-danger">Scissors</span>

</h3>

</div>

**Przycisk zainicjowania nowej gry**

To kluczowy element - służy on do uruchamiania rozgrywki.

Całemu elementowi zostaje nadane id js-newGameElement, aby móc odwołać się do niego z poziomu JavaScriptu.

Jest to ważne, ponieważ chcemy ukrywać ten element po rozpoczęciu rozgrywki. Dodatkowo, nasz przycisk ma wykonywać pewne czynności po jego kliknięciu(zdarzenie click), dlatego podpinamy do niego zdarzenie click. Lepiej jest to robić w kodzie Javascript niż poprzez atrybut onclick.

Wartość napisu będzie się zmieniać, dlatego potrzebujemy tutaj mieć również id, do którego odwołamy się przez JavaScript.

<div id="js-newGameElement" class="container">

<div class="row text-center">

<button id="js-newGameButton" class="btn btn-default">

New game

</button>

</div>

</div>

Kod Javascript:

var newGameBtn = document.getElementById('js-newGameButton');

newGameBtn.addEventListener('click', newGame);

**Wybór gracza**

Ten element zawiera przyciski wyboru gracza. Po kliknięciu gracza musimy przekazać informację, co zostało wybrane.

Każdy z elementów będzie miał podpięty listener click, który wywoła funkcję playerPick z odpowiednim parametrem.

Cały element również posiada unikalne id js-playerPickElement, aby można było go łatwo znaleźć za pomocą DOM API JavaScriptu.

<div id="js-playerPickElement" class="container">

<div class="row text-center">

<button id="js-playerPick\_rock" class="btn">

<span class="badge">

<i class="fa fa-hand-rock-o" aria-hidden="true"></i>

</span>

Rock

</button>

<button id="js-playerPick\_paper" class="btn">

<span class="badge">

<i class="fa fa-hand-paper-o" aria-hidden="true"></i>

</span>

Paper

</button>

<button id="js-playerPick\_scissors" class="btn">

<span class="badge">

<i class="fa fa-hand-scissors-o" aria-hidden="true"></i>

</span>

Scissors

</button>

</div>

</div>

Każdy z przycisków wyboru gracza posiada unikalną klasę. Należy jeszcze wskazać, co będzie działo się po ich kliknięciu.

var pickRock = document.getElementById('js-playerPick\_rock'),

pickPaper = document.getElementById('js-playerPick\_paper'),

pickScissors = document.getElementById('js-playerPick\_scissors');

pickRock.addEventListener('click', function() { playerPick('rock') });

pickPaper.addEventListener('click', function() { playerPick('paper') });

pickScissors.addEventListener('click', function() { playerPick('scissors') });

W powyższym kodzie, ustawimy listenery na każdy z przycisków. Po jego kliknięciu zostanie wywołana funkcja playerPick z parametrem reprezentującym wybór gracza.

**Tabela wyników**

W tym elemencie wyświetlamy aktualny stan gry - zarówno wybór gracza, jak i punktację oraz wynik pojedynczej rundy.

Tutaj będziemy aktualizować poszczególne elementy tabeli wyników, dlatego każdy z elementów, który będziemy aktualizować, otrzymał unikalne id.

<div id="js-resultsTableElement" class="container">

<div class="row">

<div class="col-xs-5 text-center">

<span id="js-playerPoints" class="badge">

0

</span>

<span id="js-playerName">

Player Name

</span>

</div>

<div class="col-xs-2 text-center">

VS.

</div>

<div class="col-xs-5 text-center">

<span id="js-computerPoints" class="badge">

0

</span>

Computer

</div>

</div>

<div class="row">

<div class="col-xs-5 text-center">

<span id="js-playerPick">

Player selection

</span>

</div>

<div class="col-xs-2"></div>

<div class="col-xs-5 text-center">

<span id="js-computerPick">

Computer selection

</span>

</div>

</div>

<div class="row">

<div class="col-xs-5 text-center">

<span id="js-playerResult">

Player Score

</span>

</div>

<div class="col-xs-2"></div>

<div class="col-xs-5 text-center">

<span id="js-computerResult">

Computer Score

</span>

</div>

</div>

</div>

**Czego się teraz spodziewać**

Na razie mamy stworzoną strukturę HTML + CSS (dzięki dołączonym stylom), ale nie napisaliśmy jeszcze funkcjonalności.

Chciałbym, żebyśmy zadbali jeszcze o odstępy pomiędzy kontenerami. Dlatego na kontenery nadaj odpowiednie style (np. padding).

U mnie wygląda to tak:

[](https://kodilla.com/static/bootcamp/8-13-rock-paper.png)

**Logika gry**

Po pierwsze, aktualny wynik gry oraz jej stan należy gdzieś przechowywać.

**Wartości początkowe**

Najpierw zainicjujmy wartości, których będziemy używać w grze i nadajmy im wartości początkowe.

var gameState = 'notStarted', //started // ended

player = {

name: '',

score: 0

},

computer = {

score: 0

};

Ustalamy stan gry na notStarted. Stan gry będzie nam służył do ustalenia, które kontenery należy w danej chwili wyświetlać.

Poza tym przechowujemy obiekt *player*, który będzie trzymał nazwę gracza oraz aktualny wynik, a także obiekt *computer*. Jednak ten posiada tylko właściwość *score* do trzymania wyniku, bo jego nazwa jest stała.

**Wyświetlanie elementów gry**

Każdemu z kontenerów gry nadane zostało unikalne id - czas, aby na coś się przydało :)

var newGameElem = document.getElementById('js-newGameElement'),

pickElem = document.getElementById('js-playerPickElement'),

resultsElem = document.getElementById('js-resultsTableElement');

W ten sposób tworzymy zmienne, które będą wskazywać na elementy gry, a konkretnie jej poszczególne części.

Wcześniej zdefiniowaliśmy zmienną gameState. Decydujemy, że może ona przyjąć kilka wartości - zależnie od tego, czy gra nie została jeszcze rozpoczęta, jest w trakcie czy została zakończona chcemy wyświetlić różne elementy na stronie.

Dlaczego nie stworzyć na to funkcji?

function setGameElements() {

switch(gameState) {

case 'started':

newGameElem.style.display = 'none';

pickElem.style.display = 'block';

resultsElem.style.display = 'block';

break;

case 'ended':

newGameBtn.innerText = 'Jeszcze raz';

case 'notStarted':

default:

newGameElem.style.display = 'block';

pickElem.style.display = 'none';

resultsElem.style.display = 'none';

}

}

Zależnie od stanu zmiennej gameState zostaną wykonane różne polecenia.

Kiedy gra jest "started", nie chcemy widzieć kontenera do rozpoczęcia gry, a jedynie możliwość wyboru przedmiotu i tabele rezultatów.

Kiedy gra będzie zakończona, nasz przycisk rozpoczynania gry ma zmienić napis na "Play again", a następnie być wyświetlony i ukryć pozostałe elementy gry (zauważ brak instrukcji break;).

W przypadku kiedy gra jest "notStarted" (dzieje się tak tylko przed rozpoczęciem gry), przycisk ma zachować stan "Nowa gra" i być widoczny, a pozostałe elementy ukryte.

Skoro funkcja już jest, pozostaje nam tylko ją wywołać.

setGameElements();

Sprawdź, jak zachowuje się gra, kiedy nadasz zmiennej gameState różne wartości.

**Rozpoczęcie gry**

Teraz należy zdefiniować funkcję newGame, która będzie odpowiadać za rozpoczęcie każdej gry.

var playerPointsElem = document.getElementById('js-playerPoints'),

playerNameElem = document.getElementById('js-playerName'),

computerPointsElem = document.getElementById('js-computerPoints');

Na początek zdefiniujmy zmienne odnoszące się do tych elementów na stronie, które będziemy aktualizować przed rozpoczęciem rozgrywki. Po kolei jest to punktacja gracza, imię gracza wyświetlane na stronie oraz punktacja komputera.

Dalej definiujemy funkcję, która będzie uruchamiana po wciśnięciu przycisku "New Game" / "Play Again"

function newGame() {

player.name = prompt('Please enter your name', 'imię gracza');

if (player.name) {

player.score = computer.score = 0;

gameState = 'started';

setGameElements();

playerNameElem.innerHTML = player.name;

// setGamePoints(); // This function has not been created yet

}

}

Najpierw pobieramy imię gracza i zapisujemy je w naszym obiekcie.

Następnie sprawdzamy, czy użytkownik podał imię. Jeśli pole było puste lub zostało kliknięte "Anuluj", instrukcja warunkowa nie zostanie wykonana, a zatem gra się nie rozpocznie.

Jeśli gracz prawidłowo podał imię i kliknął OK, wyniki zostają wyzerowane, status gry zmieniony, a następnie wywołana funkcja, która ustala, co w tym momencie powinno być widoczne na ekranie gry.

Ostatnia instrukcja służy do wpisania imienia gracza w tablicy wyników na stronie.

Zdefiniowaliśmy także funkcję do ustalania wyniku, jednak na razie pozostawmy ją zakomentowaną. Punkty w obiektach zostały wyzerowane, jednak będziemy musieli zadbać, aby tak stało się również w elementach strony.

**Wybór gracza**

Zdefiniujmy teraz funkcję, która odpowiada za pobranie wyboru gracza.

function playerPick(playerPick) {

console.log(playerPick);

}

Na razie wybór wyświetlamy jedynie w konsoli. Funkcja playerPick wywołana zostanie za każdym razem, kiedy użytkownik kliknie przycisk, a więc połowa za nami.

**Losowanie wyboru komputera**

Kiedy określiliśmy już, co wybrał gracz, trzeba napisać funkcję losującą wybór komputera.

Jak tego dokonać? Przy użyciu obiektu Math. Zawiera on w sobie, jak sama nazwa wskazuje, metody matematyczne. Pierwsza, jaką poznamy to Math.random.

Jeżeli przypiszemy ją do zmiennej, metoda ta zwraca nam losową liczbę zmiennoprzecinkową:

var x = Math.random();

to x może wynosić przykładowo:

0.9200883881643283

Jednak to tylko liczba, a musimy wybrać losowo jedną z trzech opcji.

By to zrobić, nakażemy komputerowi wylosować jedną liczbę, a następnie pomnożymy ją przez 3. Potem skorzystamy z kolejnej metody matematycznej: Math.floor, która zaokrągla liczby zmiennoprzecinkowe w dół, do liczby całkowitej.

Poniższy zapis zwróci nam losową liczbę całkowitą z przedziału od 0 do 2 (czyli 0, 1 lub 2):

Math.floor(Math.random()\*3)

Jest to bardzo popularna metoda na generowanie losowych wyborów.

Teraz każdej z liczb przypiszemy odpowiednio kamień, papier lub nożyce. Ciekawym rozwiązaniem będzie zwrócenie wybranego elementu tablicy.

function getComputerPick() {

var possiblePicks = ['rock', 'paper', 'scissors'];

return possiblePicks[Math.floor(Math.random()\*3)];

}

Mamy wybór gracza oraz komputera - skoro już potrafimy otrzymać te dwie rzeczy, umieśćmy ten wybór na stronie.

var playerPickElem = document.getElementById('js-playerPick'),

computerPickElem = document.getElementById('js-computerPick'),

playerResultElem = document.getElementById('js-playerResult'),

computerResultElem = document.getElementById('js-computerResult');

function playerPick(playerPick) {

var computerPick = getComputerPick();

playerPickElem.innerHTML = playerPick;

computerPickElem.innerHTML = computerPick;

}

Jeśli w tej chwili uruchomisz grę, powinieneś móc ją rozpocząć, a po wybraniu przycisku rock, paper lub scissors zobaczyć w tabeli wybór gracza oraz wybór komputera.

**Logika gry i przyznawanie punktów**

Kamień pokonuje nożyce, nożyce papier, a papier kamień. Remis to oczywiście remis :)

Moglibyśmy wyszczególnić tutaj wszystkie możliwe sytuacje za pomocą instrukcji warunkowych. Co jeśli gracz wybrał kamień, a komputer nożyce lub papier? A może jest remis, bo obaj gracze dokonali tego samego wyboru?

Bądźmy *spryciarzami*.

Na początek załóżmy, że to my wygraliśmy, a potem sprawdźmy czy to prawda.

function checkRoundWinner(playerPick, computerPick) {

playerResultElem.innerHTML = computerResultElem.innerHTML = '';

var winnerIs = 'player';

if (playerPick == computerPick) {

winnerIs = 'noone'; // remis

} else if (

(computerPick == 'rock' && playerPick == 'scissors') ||

(computerPick == 'scissors' && playerPick == 'paper') ||

(computerPick == 'paper' && playerPick == 'rock')) {

winnerIs = 'computer';

}

if (winnerIs == 'player') {

playerResultElem.innerHTML = "Win!";

player.score++;

} else if (winnerIs == 'computer') {

computerResultElem.innerHTML = "Win!";

computer.score++;

}

}

Na początku usuwamy wyświetlany tekst o wygranej któregoś z graczy. Jeszcze nie wiadomo, kto wygrał, będziemy to dopiero sprawdzać.

Funkcja otrzymuje wybory obu graczy. Na początek zakładamy, że to my wygraliśmy rundę - dzięki temu nie musimy tego sprawdzać, a jedynie potwierdzić.

Jeśli był remis, bo obaj gracze wybrali to samo, to wartość winnerIs zostanie nadpisana i zwrócona.

Jeśli nie było remisu, to sprawdzamy, czy komputer trafił w którąś z możliwości jego wygranej. Jeśli tak, to wartość winnerIs zostanie nadpisana i zwrócona.

Jeśli te sytuacje nie miały miejsca, to znaczy, że wygrana musi być nasza.

Na koniec funkcji dodajemy punkty zwycięzcy i stosowną informację w tabeli wyników.

Nasza funkcja powinna wywoływać się za każdym wyborem gracza. W tej chwili powinna wyglądać następująco.

function playerPick(playerPick) {

var computerPick = getComputerPick();

playerPickElem.innerHTML = playerPick;

computerPickElem.innerHTML = computerPick;

checkRoundWinner(playerPick, computerPick);

}

**Aktualizacja wyniku**

Teraz dodajemy wartości. Wynik gry znajduje się wewnątrz obiektów player oraz computer- brakuje tylko wyświetlania go na stronie.

Stwórzmy funkcję, która będzie za to odpowiedzialna.

function setGamePoints() {

playerPointsElem.innerHTML = player.score;

computerPointsElem.innerHTML = computer.score;

}

Ta krótka i prosta funkcja aktualizuje stan wyniku na elementach strony, pamiętasz? Dodawaliśmy ją w funkcji newGame() - proszę, odkomentuj ją teraz.

**Zakończenie rozgrywki**

Przyda nam się jeszcze funkcja, która sprawdzi, czy któryś z graczy zdobył 10 punktów.

Jednak to zostawiam już w pełni Tobie. Powodzenia!

**Potencjalne problemy**

Jeśli napotkasz jakiś problem, to po pierwsze, sprawdź, czy masz jakieś błędy w konsoli.

Jeśli chcesz sprawdzić, czy jakaś funkcja się uruchamia, wrzuć tam console.log - być może zdarzyło Ci się zapomnieć o jej wywołaniu.

Gra nie działa tak, jak powinna? Ponownie użyj console.log i sprawdź wartości, jakie w danej chwili posiadają zmienne.

**Podsumowanie rozdziału**

Uff, udało Ci się! To solidny kawał wiedzy. Z pewnością zanim będziesz mógł pisać skrypty z pamięci, przyda Ci się jeszcze sporo praktyki - na to potrzeba czasu.

Jednak nie ma wątpliwości, że stworzyłeś swoją pierwszą grę - jestem z Ciebie dumny! To zadanie z pewnością przyda Ci się w portfolio, szczególnie jeśli będziesz umiał powiedzieć, co się dzieje w poszczególnych fazach gry.

**Gratulacje!**

**Zadanie: Kamień, papier, nożyce!**

Dopisz kod odpowiadający za zakończenie rozgrywki. Wyświetl informację o zwycięzcy.

Stwórz nowe repozytorium i wyślij tam swoje zadanie, a link do niego wklej poniżej. Pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/108241)

Początek formularza

 Wyslij link

Dół formularza

**9.8. Dodatkowy projekt dla ambitnych**

Kolejny dodatkowy projekt dla ambitnych :)

Zasady są nadal takie same:

* Projekt jest NIEOBOWIĄZKOWY
* Wykonując go, zdajesz się na umiejętności które zdobyłeś/aś w czasie kursu
* Możesz korzystać jedynie z pomocy społeczności na czacie.

Jak wiadomo, chodzi o nabranie praktyki przy cięciu szablonów. Czyli zadaniu którym Front-endowcy zajmują się najczęściej.

Jeżeli nie chcesz wykonywać tego projektu to po prostu kliknij w przycisk i o nim zapomnij ;)

Jednak jeżeli chcesz poćwiczyć to Twoim zadaniem jest:

1. Utworzenie nowego projektu na GitHubie.
2. Zakodowanie layoutu na podstawie grafiki.
3. Uzupełnienie treści, mogą być sztuczne (wypełniacze).
4. Zachowanie dobrych praktyk przedstawionych w dotychczasowych treściach.
5. Wykorzystanie Bootstrapa.
6. Przystosowanie strony do odpowiedniego wyświetlania na urządzeniach mobilnych.

Poniżej znajdziesz link do grafiki do pobrania w formacie .png:  
[Link do grafiki](https://kodilla.com/static/bootcamp/9-8-project.png)

Poniżej znajdziesz link do grafiki do pobrania w formacie .psd:  
[Link do pliku .psd](https://uploads.kodilla.com/bootcamp-fe/project-06-FE-dodatkowe-zadanie-m9.psd)

W submodule 3.4 znajdziesz tutoriale, które pomogą Ci odwzorować szablon korzystając z plików .psd.

Powodzenia! :)

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

Zapoznałe(a)m się!