Kurs Front-End Developer

JavaScript

KOMENTARZE (I-***)

W JavaScript można stosować dwa rodzaje komentarzy – `wierszowe` i `blokowe`.

`Komentarz blokowy` rozpoczyna się od znaków /* i kończy znakami */. Wszystko co znajduje się pomiędzy tymi znakami jest pomijane przy kompilowaniu kodu. Komentarzy tych nie można zagnieżdżać, ale można stosować wewnątrz nich komentarze liniowe

Visual Studio Code:
`Shift + Alt + A (Mac)` lub `Shift + Alt + A (Win & Linux)`

`Komentarz wierszowy` (liniowy) zaczyna się od znaków // i obowiązuje do końca danej linii skryptu. Wszystko co znajduje się po tych znakach, aż do końca bieżącej linii jest pomijane podczas kompilowania kodu

`Cmd/CTRL + /`

ZMIENNE I STAŁE (2-***)

`Zmienna` i `stałe` przechowują dane, do których możemy się odwołać poprzez ich `nazwę`.

Zmienne tworzone są za pomocą słowa kluczowego `let` lub `var`, po którym następuje nazwa zmiennej.

let nazwaZmiennej; //deklaracja zmiennej ES6
var nazwaZmiennej; //deklaracja zmiennej ES5

Zmiennej można przypisać wartość za pomocą operatora przypisania czyli znaku = (równa się):
let nazwaZmiennej = `wartoscZmiennej;` *let number = 10;*

Zmienną można nadpisać
nazwaZmiennej = `nowaWartoscZmiennej;` *number = 15;*

Stałe definiujemy za pomocą słowa kluczowego `const`, po którym następuje nazwa stałej.
const nazwaStalej = wartoscStalej; *const numberOfMonths = 12;*

Stałe w JavaScript nie mogą zostać napisane w toku wykonywania się skryptu.
~~*nazwaStalej = nowaWartoscStalej;*~~ *// nie można wykonać takiej operacji*

ZMIENNE I STAŁE (2-***)

`Nazewnictwo zmiennych i stałych` - zasady:

- nazwa zmiennej `powinna zaczynać się od małej litery`
- kolejne wyrazy pisane są łącznie, rozpoczynając każdy następny wielką literą (prócz pierwszego) – notacja `camelCase`,
- nazwa zmiennej `nie może się zaczynać od cyfry (0-9)`,
- nazwa zmiennej `nie może zawierać spacji`,
- nazwa zmiennej `nie może zawierać polskich liter`,
- nazwą zmiennej `nie może być słowo kluczowe zarezerwowane przez JavaScript` czyli takie słowo które ma już specjalne znaczenie w JS (np. *this* czy *let*).

Nazwy zmiennych powinny być tworzone tak, aby opisywał jakie dane przechowują.

Należy też pamiętać o tym, że w JS istnieje rozróżnienie pomiędzy dużymi i małymi literami
let number; *oraz* ~~*let Number;*~~ - to dwie różne zmienne

FUNKCJE (3-***)

`Funkcje` są to wydzielone bloki kodu przeznaczone do wykonywania konkretnych zadań.

Tworzy się je przy użyciu słowa kluczowego `function`.

Tworzenie funkcji zwiększa przejrzystość kodu i ułatwia programowanie oraz pozwala `wielokrotnie wykonywać ten sam zestaw instrukcji` bez konieczności każdorazowego pisania tego samego kodu.

Funkcja jest wywoływana przez inną część skryptu, a w momencie jej wywołania zostaje wykonywany kod w niej zawarty.

FUNKCJE (3-***)

Ogólna `deklaracja` funkcji jest postaci:

```
function nazwaFunkcji() {
    // kod funkcji
}
nazwaFunkcji();   // wywołanie funkcji
```

`nazwaFunkcji` – dowolna nazwa która powinna spełniać takie same wymogi jak nazwy zmiennych

Funkcję możemy stworzyć także za pomocą `wyrażenia funkcyjnego`, do którego przypisujemy funkcję anonimową.

```
const nazwaFunkcji = function() {
    // kod funkcji anonimowej
}
nazwaFunkcji();   // wywołanie funkcji
```

FUNKCJE (3-***)

Funkcją można przekazywać `parametry`, czyli wartości (dane), które mogą wpływać na działanie funkcji lub też być przez funkcję przetwarzane.

Parametry przekazuje się wypisując je między nawiasami występującymi po nazwie funkcji, poszczególne parametry oddzielamy od siebie przecinkiem:

```
function nazwaFunkcji(parametr1, parametr2, parametr3) {
    // kod funkcji
}

// wywołanie funkcji
nazwaFunkcji(wartoscParametru1, wartoscParametru2, wartoscParametru3);
```

FUNKCJE (3-***)

Dzięki zastosowaniu instrukcji `return` możemy nakazać funkcji zwracanie określonej wartości.

Instrukcja ta równocześnie `przerywa dalsze działanie funkcji` i powoduje zwrócenie wartości występującej po `return`.

Poprzez słowo kluczowe `return` funkcja zwróci wartość, która będzie mogła być wykorzystana w dalszej części skryptu.

```
function addNumbers(parametr1, parametr2) {
        let result = parametr1 + parametr2;
        return result;
}

let sum = addNumbers(1,2); // do zmiennej sum zostanie przypisana wartość 3
```

FUNKCJE (3-***)

Standard ES6 wprowadził możliwość definiowania `funkcji strzałkowych`.

```
const nazwaFunkcji = () => {
        // kod funkcji
}
nazwaFunkcji(); // wywołanie funkcji
```

Funkcja z parametrami:

```
const nazwaFunkcji = (parametr1, parametr2) => {
        // kod funkcji
}
nazwaFunkcji(parametr1, parametr2); // wywołanie funkcji
```

Zapis skrócony dla funkcji, która ma tylko jeden parametr oraz zwraca wynik – można pominąć nawias przy definicji funkcji oraz nawiasy klamrowe stanowiące ciało , funkcji oraz instrukcję return

```
const nazwaFunkcji = a =>  a*a;
```

ZASIĘG ZMIENNYCH (4-***)

Gdy pracujemy z funkcjami i zmiennymi mamy również do czynienia z `pojęciem zasięgu zmiennych.`

Zasięg możemy określić jako miejsca, w których zmienna jest widoczna i można się do niej bezpośrednio odwoływać.

W JavaScript możemy korzystać ze `zmiennych globalnych` oraz `zmiennych lokalnych.` `Zmienne globalne` są dostępne dla całego skryptu tzn. dla wszystkich funkcji, metod, operacji jaki wykonujemy w skrypcie. `Zmienne lokalne` są dostępne tylko np. we wnętrzu danej funkcji lub bloku kodu.

```
function addNumbers(parametr1, parametr2) {
        let result = parametr1 + parametr2; // zmienna lokalna
        return result;
}

let sum = addNumbers(1,2); // zmienna globalna
```

TYPY DANYCH (5-***)

W JavaScript występuje kilka typów danych, które ogólnie dzielą się na typy `proste` i `referencyjne`.

`Typy proste` służą do zapisywania prostych danych takich jak:

- `liczb` - typ liczbowy
- `łańcuchów znaków (tekstu)` - typ łańcuchowy
- `wartości prawda/fałsz` - typ logiczny
- `null` i `undefined` - typy specjalne

`Typy referencyjne` służą do przechowywania obiektów, dlatego mówimy o nich również jako o `typach złożonych`. Nie przechowują bezpośrednio wartości, ale wskazują na miejsce w pamięci, gdzie dane obiektu są przechowywane. Podstawowe rodzaje typów referencyjnych to:

- `tablice`
- `obiekty`

Typ zmiennej można sprawdzić poprzez poprzedzenie jej nazwy słowem `typeof`

```
typeof nazwaZmiennej; //zwróci typ zmiennej
```

TYPY DANYCH (5-***)

`Typ liczbowy (number)` – służy do reprezentacji liczbowej, np.

```
let number = 10;
```

Możliwe formaty zapisu:

- `zapis liczb całkowitych i ułamkowych, np. 0, 1, -2, 3.0, 3.14, -6.28.` - opcjonalnie podajemy znak liczby, potem część całkowitą i opcjonalnie część ułamkową oddzieloną znakiem kropki.
- `zapis liczby systemem szesnastkowym.` - zapis takiej liczby rozpoczynamy od 0x lub 0X, po czym piszemy sekwencję znaków 0-9a-fA-F, np. 0x0, 0X1, 0xFF, -0xAB.
- `zapis notacją wykładniczą, np. 1e3, 314e-2, 2.718e0.` - zapis naukowy rozszerza standardową notację o część zawierającą e lub E oraz liczbę całkowitą będącą wykładnikiem (z opcjonalnym znakiem + lub -).
- `zapis systemem ósemkowym` – zapis rozpoczyna się od cyfry zero.

TYPY DANYCH (5-***)

`Typ łańcuchowy (string)` - wartość tego typu jest sekwencją zera lub więcej znaków umieszczonych pomiędzy dwoma cudzysłowami lub apostrofami, np.

```
let sentence = 'Ola ma kota';
```

Ciąg może zawierać sekwencje specjalne:

- \n - nowy wiersz (ang. *new line*)
- \' - cudzysłów (ang. *double quote*)
- \" - apostrof (ang. *single quote*)
- \\ - lewy ukośnik (ang. *backslash*)

Standard `ES6` powala również na definiowanie stringów w znakach `` `(backtick)`, które powalają na osadzanie wartości innych zmiennych w tekście za pomocą znaków `${}` – jest to tak zwana `interpolacja stringów`

```
let  age = 12;
let sentence = `Ola ma ${age} lat`; //Ola ma 12 lat
```

TYPY DANYCH (5-***)

`Typ logiczny (boolean)` - pozwala na określenie dwóch wartości logicznych: prawda i fałsz. Wartość prawda jest w języku JavaScript reprezentowana przez słowo `true`, natomiast wartość fałsz — przez słowo `false`, np.

```
let letBol = true;
```

`Typy specjalne:`
`undefined` – oznacza, że dana zmienna nie została zdefiniowana

`null` - podobnie jak w innych językach programowania, oznacza nic. Został pomyślany jako wyznacznik braku referencji do obiektu. W praktyce, z `null` spotkamy się używając funkcji wyszukujących element w dokumencie, np.

```
let element = document.getElementById( 'id-elementu' );
if ( element !== null ) {
    // logika programu
}
```

TABLICE (6-***)

`Tablice` - to struktury danych pozwalające na przechowywanie uporządkowanego zbioru elementów.

Aby utworzyć tablicę, deklarujemy zmienną i przypisujemy do niej w kwadratowych nawiasach wartości rozdzielone przecinkami np.

```
let names = [ 'Krystian', 'Ania', 'Adam' ];
```

Odczyt zawartości danej komórki osiągamy poprzez podanie jej indeksu w nawiasie kwadratowym:

```
nazwaTablicy[ indeksKomorki ];
```

np.

```
names[ 2 ];
```

`Tablice są indeksowane od 0`, tak więc pierwszy element tablicy ma index - 0, drugi - 1, trzeci - 2 itd.

TABLICE (6-***)

Aby dodać nową wartość do tablicy `ustawiamy nową wartość w odpowiednim indeksie tablicy` lub korzystamy z `metody push('Nowy element')`, która dodaje nowy element na końcu tablicy i zwraca jej długość:

```
let names = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę
names[3] = 'Piotrek';                               // dodaj wartość do tablicy
names[4] = 'Grzegorz';                              // dodaj wartość do tablicy

console.log(names[3] + ' i ' + names[4] );  // konsola wypisze się 'Piotrek i Grzegorz'

names.push( 'Michał' );                 // dodaj wartość na koniec tablicy i zwraca jej długość
console.log(names[5] );                 // wyloguje Michał
```

TABLICE (6-***)

Odwrotnie do metody *push()* działa `metoda pop()`, która `usuwa ostatni element z tablicy` po czym go zwraca.

```
let names = [ 'Marcin', 'Ania', 'Agnieszka' ]; //stwórz tablicę
names.pop();                                    // usuwa ostatni element i zwraca jego wartość
console.log( names );                           // wyloguje się 'Marcin,Ania'
```

`Metoda unshift()` wstawia nowy element do tablicy na jej początku, po czym zwraca nową długość tablicy.

```
let names = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę
names.unshift( 'Piotrek', 'Paweł' );    //dodaje nowe elementy i zwraca długość tablicy
console.log( names );                   //wyloguje się 'Piotrek, Paweł, Marcin,Ania,Agnieszka'
```

TABLICE (6-***)

`Metoda shift()` usuwa pierwszy element z tablicy i go zwraca.

```
let names = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę
names.shift();                                      // usuwa pierwszy element i go zwróci
console.log( names );                               // wyloguje się 'Ania,Agnieszka'
```

Każda tablica udostępnia nam właściwość `length`, dzięki której można określić długość tablicy (ilość elementów).

```
let names = [ 'Marcin', 'Ania', 'Agnieszka' ];      // stwórz tablicę
console.log( names.length );                        // 3
```

TABLICE (6-***)

`Metoda join()` służy do łączenia kolejnych elementów w jeden tekst.

Opcjonalnym parametrem tej metody jest znak, który będzie oddzielał kolejne elementy w utworzonym tekście. Jeżeli go nie podamy będzie użyty domyślny znak przecinka.

```
let names = [ 'Marcin', 'Ania', 'Agnieszka' ]; //stwórz tablicę

console.log( names.join() );          // wyloguje 'Marcin,Ania,Agnieszka'
console.log( names.join( ' - ' ) );   // wyloguje 'Marcin - Ania - Agnieszka'
console.log( names.join( ' + ' ) );   // wyloguje 'Marcin + Ania + Agnieszka'
```

TABLICE (6-***)

Dzięki `metodzie reverse()` można odwrócić elementy tablicy.

```
let names = [ 'Marcin', 'Ania', 'Agnieszka' ]; //stwórz tablicę

names.reverse();              // odwrócenie
console.log( names );         // wypisze się ' Agnieszka,Ania, Marcin'
```

`Metoda sort()` służy do sortowania tablicy.

```
let names = [ 'Marcin', 'Ania', 'Piotrek', 'Grześ' ];
names.sort();                 // podstawowa wersja metody
console.log( names );         // wypisze się 'Ania, Grześ, Marcin, Piotrek'
```

OBIEKTY (7-***)

`Obiekty`, podobnie jak tablice, służą do przechowywania zbiorów danych. Różni je to, że w tablicach dane są przypisane do kluczy o wartościach liczbowych (0, 1, 2 itd.) W obiektach `tworzymy klucze samodzielnie`, tak żeby ich nazwa `odpowiadała przechowywanej w niej zawartości`. Dodatkowo dla obiektów możemy samodzielnie definiować ich metody.

```
//Obiekt w ES5
let person = {
    name:    'Marcin',
    height:    184,
    print :    function() {
        console.log( this.name );
    }
}
```

```
//Obiekt w ES6
let person = {
    name:    'Marcin',
    height:    184,
    print() {
        console.log( this.name );
    }
}
```

Wnioski, z powyższej konstrukcji:

- zmienna, która przechowuje obiekt nazywa się `instancją/obiektem`
- obiekt definiuje się za pomocą `nawiasów {}`
- elementy składowe obiektu (pola) `rozdzielone są przecinkiem`
- pary `klucz-wartość są rozdzielone dwukropkiem` – takie pary to są to właściwości obiektu
- obiekt może posiadać `metody`, są to działania które mogą być wykonywane na obiektach - są to `wewnętrzne funkcje` obiektów.

OBIEKTY (7-***)

Dostęp do właściwości obiektu:

nazwaObiektu.kluczWlasnosci; lub *nazwaObiektu['kluczWlasnosci'];*

np. *person.name;* lub *person['name'];*

Dostęp do metod obiektu:

nazwaObiektu.nazwaMetody(); lub *nazwaObiektu['nazwaMetody']();*

np. *person.print();* lub *person['print']();*

Aby odwołać się do danego obiektu z jego wnętrza stosujemy `słowo kluczowe this`, np. *this.name;*.

Dodawanie właściwości:

```
let person = {
    name:       'Marcin',                   // właściwość obiektu
    height:     184,
    print() { console.log( this.name ); }   // metoda obiektu
}

person.weight = 73;                         // dodawanie właściwości
```

KLASY (7-***)

W sytuacji, gdy chcemy utworzyć kilka obiektów, które mają określone właściwości i metody to wykorzystamy do tego tak zwaną `klasę obiektu`.

Klasa to `szablon`, który definiuje jak będą wyglądać i jak będą się zachowywać tworzone w oparciu o nią obiekty.

Do stworzenia klasy obiektu wykorzystujemy słowo kluczowe `function (ES5)` lub słowo kluczowe `class (ES6)`. Tworząc klasę słowem kluczowym `class` potrzebny jest także konstruktor – czyli odpowiednia metoda `constructor()`, która będzie budować nam obiekty tej klasy. Nazwę klasy definiujemy wielką literą.

Pojedynczy obiekt stworzony na podstawie klasy, to instancja klasy. Tworzy się go poprzez słowo kluczowe `new` i nazwę klasy.

```
let person = new Person();
```

KLASY (7-***)

```
//Tworzymy klasę obiektu Person ES5
function Person(name, surname) {
    this.name = name;
    this.surname = surname;
    this.printInfo = function() {
        console.log('Imię: ' + this.name+ ', ' + 'Nazwisko: ' + this.surname);
    }
}
```

```
//Tworzymy klasę obiektu Person ES6
class Person {
    constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }

    printInfo() {
        console.log( `Imię: ${this.name}, Nazwisko ${this.surname}` );
    }
}
```

```
let krystian = new Person('Krystian', 'Dziopa'); // stwórz nową instancję obiektu Person

krystian.printInfo(); //Wyloguje 'Imię: Krystian, Nazwisko: Dziopa

let lukasz = new Person('Łukasz', 'Badocha'); // stwórz nową instancję obiektu Person

lukasz.printInfo(); //Wyloguje 'Imię: Łukasz, Nazwisko: Badocha
```

OBIEKT MATH (8-***)

`Obiekt Math` zawiera stałe matematyczne oraz metody pozwalające na wykonywanie różnych operacji matematycznych, takich jak pierwiastkowanie, potęgowanie itp.

Jest to obiekt wbudowany, co oznacza, że można z niego korzystać bezpośrednio, bez wywoływania nowej instancji.

Stałe matematyczne i metody dostępne dzięki obiektowi `Math`:

Math.E	-	zwraca stałą Eulera, która wynosi ok. 2.71
Math.LN10	-	zwraca logarytm z dziesięciu, tj. ok. 2.30
Math.PI	-	zwraca wartość liczby Pi, czyli ok. 3.14
Math.SQRT2	-	zwraca pierwiastek kwadratowy z 2, czyli ok. 1.41

Math.cos(iloscStopni) - zwraca cosinus kąta
Math.pow(podstawa, wykladnik) - zwraca liczbę podniesioną do potęgi
Math.random() - zwraca losową liczbę z zakresu od 0 do 1

```
console.log( 'PI = ' + Math.PI );
console.log( 'cos(0) = ' + Math.cos(0) );
```

OPERATORY (9-***)

Na zmiennych można wykonywać róże operacje za pomocą `operatorów`. Dzielą się na cztery rodzaje: `arytmetyczne, porównania, przypisania, logiczne.`

Operatory `arytmetyczne` – pozwalają na podstawowe operacje matematyczne:

- `+` - dodawanie
- `-` - odejmowanie
- `*` - mnożenie
- `/` - dzielenie
- `%` - modulo – różnica z dzielenie
- `++` - inkrementacja (powiększ o 1)
- `--` - dekrementacja (zmniejsz o 1)

Możemy mówić o postinkrementacji/postdekrementacji i preinkrementacji/predekrementacji :

++nazwaZmiennej; //preinkrementacja – najpierw dodaj jednostkę, potem zwróć nową wartość

nazwaZmiennej++; //postinkrementacja – najpierw zwróć nową wartość, potem dodaj jednostkę

OPERATORY (9-***)

Operatory `przypisania` - powodują przypisanie wartości argumentu znajdującego się z prawej strony operatora argumentowi znajdującemu się z lewej strony.

- `=` - przypisz wartość
- `+=` - dodaj do zmiennej wartość i przypisz do niej sumę
- `-=` - odejmij od zmiennej wartość i przypisz do niej różnicę
- `*=` - pomnóż zmienną i przypisz do niej iloczyn
- `/=` - podziel zmienną i przypisz do niej iloraz
- `%=` - podziel zmienną i przypisz do niej różnicę z dzielenia

```
let number = 10;  //Przypisz 10
number += 5;  //Do 10 dodaj 5 i przypisz sumę
console.log( numer ); //Wyloguje 15
```

OPERATORY (9-***)

Operatory `porównania` - służą do porównywania argumentów. Wynikiem ich działania jest wartość logiczna `true` lub `false`, czyli prawda lub fałsz.

Operatory tego typu najczęściej wykorzystywane są w połączeniu z instrukcjami warunkowymi.

- `==` - równe wartości
- `!=` - różne wartości
- `===` - równe wartości i typ danych
- `!==` - różne wartości lub/i typ danych
- `>` - większe od
- `<` - mniejsze od
- `>=` - większe lub równe
- `<=` - mniejsze lub równe

OPERATORY (9-***)

Operatory `logiczne` - za pomocą operatorów logicznych możemy łączyć kilka porównań w jedną całość. Można je wykonywać na argumentach, które posiadają wartość logiczną: `true` lub `false`. Wynikiem takiej operacji jest wartość `true` lub `false`.

`Iloczyn logiczny (and) – &&` - wynikiem iloczynu logicznego jest wartość true, wtedy i tylko wtedy, kiedy oba argumenty mają wartość true. W każdym innym przypadku wynikiem jest false.

```
(2<3 && 3>1) //true
(10<3 && 3>1) //false
```

`Suma logiczna (or) – ||` - wynikiem sumy logicznej jest wartość false, wtedy i tylko wtedy, kiedy oba argumenty mają wartość false. W każdym innym przypadku wynikiem jest true.

```
(2>3 || 3>1) //true
(10<3 || 3<1) //false
```

`Negacja logiczna (not) – !` - zmieniamy wynik operacji logicznej na przeciwną, czyli jeśli argument miał wartość true, będzie miał wartość false i odwrotnie, jeśli miał wartość false, będzie miał wartość true.

```
(!(10<3 || 3<1)) //true
(!true) //false
```

OPERATORY (9-***)

`Operator warunkowy (ternary)` pozwala na ustalenie wartości wyrażenia w zależności od prawdziwości danego warunku. Ma on postać:

```
warunek ? wartość1 : wartość2
```

która oznacza: jeśli warunek jest prawdziwy, podstaw za wartość całego wyrażenia wartość1, a w przeciwnym razie za wartość wyrażenia podstaw wartość2, np.

```
let number = 100;
let wynik = ( number < 0 ) ? -1 : 1;
console.log( wynik );      // Ponieważ 100 jest mniejsze od zera do zmiennej przypisane zostało 1
```

INSTRUKCJE WARUNKOWE (10-***)

`Instrukcja warunkowa` wykonuje wybrany kod, w zależności czy wartość danego wyrażenia jest `prawdą (true)` czy `fałszem (false)`.

`Instrukcja if` ma kilka postaci, najprostsza z nich to:

```
if ( warunek ) {
        // instrukcje do wykonania jeśli warunek jest spełniony
}
```

Instrukcja `if` sprawdza dany warunek i w zależności od tego czy zwróci `true` lub `false` wykona lub nie wykona sekcję kodu zawartą w klamrach, np.

```
let number = 1;
if ( number == 1 ) {
        // instrukcja wykona się, bo prawdą jest to że zmienna number jest równa 1
        console.log( 'Liczba równa się 1' );
}
```

INSTRUKCJE WARUNKOWE (10-***)

`Instrukcja if-else` - poprzez dodanie do instrukcji `if` bloku `else` możemy sprawdzić przeciwieństwo warunku `if`

```
if ( warunek ) {
        // instrukcje do wykonania jeśli warunek jest spełniony
} else {
        // instrukcje do wykonania jeśli warunek nie jest spełniony
}
```

```
let number = -1;
if ( liczba > 0 ) {
        console.log ( 'Wartość zmiennej liczba jest większa od 0.' );
} else {
         // Wykona się ta instrukcja, bo -1  nie jest większe od 0
        console.log ( 'Wartość zmiennej liczba nie jest większa od 0.' );
}
```

INSTRUKCJE WARUNKOWE (10-***)

`Instrukcja else if` - trzecia wersja instrukcji `if` pozwala na badanie wielu warunków. Po bloku `if` może wystąpić wiele dodatkowych bloków `else if`

```
if ( warunek1 ) {
        // instrukcje1
} else if ( warunek2 ) {
        // instrukcje2
}
...
else if ( warunekN ) {
        // instrukcjeN
} else {
        // instrukcjeM
}
```

Co oznacza: jeżeli `warunek1` jest prawdziwy, to zostaną wykonane `instrukcje1` w przeciwnym razie, jeżeli jest prawdziwy `warunek2`, to zostaną wykonane `instrukcje2` w przeciwnym razie, jeśli jest prawdziwy `warunek3`, to zostaną wykonane `instrukcje3`, itd. Jeżeli żaden z warunków nie będzie prawdziwy, to zostaną wykonane `instrukcjeM`.

Ostatni blok `else` jest jednak opcjonalny i nie musi być stosowany.

INSTRUKCJE WARUNKOWE (10-***)

`Instrukcja switch` jest kolejnym sposobem testowania warunków działającym na zasadzie przyrównania wyniku do podanych przypadków.

Pozwala w wygodny sposób sprawdzić ciąg warunków i wykonać różne instrukcje w zależności od wyników porównywania.

```
switch ( wyrażenie ) {
        case przypadek1:
                // fragment wykonywany, gdy rezultat wyrażenia
                jest równy rezultat1 - potrzebuje break;
        break;

        case przypadek2:
                // fragment wykonywany, gdy rezultat wyrażenia
                jest równy rezultat2 - potrzebuje break;
        break;
        …
        default:
                //fragment wykonywany gdy, powyższe rezultaty
                nie są równe rezultatowi wyrażenia
}
```

Co oznacza:

- sprawdź wartość wyrażenia `wyrażenie`, jeśli wynikiem jest `przypadek1`, to wykonaj intrukcje i przerwij wykonywanie bloku `switch` (przerwanie jest wykonywane przez instrukcję `break`);
- jeśli wynikiem jest `przypadek2`, to wykonaj dla tego przypadku itd.
- jeśli nie zachodzi żaden z wymienionych przypadków, wykonaj instrukcję domyślną i zakończ blok `switch`
- blok `default` jest jednak opcjonalny i może zostać pominięty.

PĘTLE (II-***)

`Pętle` w programowaniu pozwalają nam wykonywać dane operacje wielokrotnie do momentu spełnienia się określnego warunku.

`Pętla for` - postać ogólna:

```
for ( wyrażenie początkowe ; wyrażenie warunkowe ; wyrażenie modyfikujące ) {
        // instrukcje do wykonania
}
```

`wyrażenie początkowe` jest stosowane do zainicjalizowania zmiennej używanej jako licznik liczby wykonań pętli

`wyrażenie warunkowe` określa warunek, jaki musi być spełniony, aby dokonać kolejnego przejścia w pętli

`wyrażenie modyfikujące` jest zwykle używane do modyfikacji zmiennej będącej licznikiem

```
for(let i = 0; i<5; i++) {
        // pętla wykona się  5 razy
}
```

PĘTLE (II-***)

`Pętla forEach` - pętla pozwala wykonywać iteracje na tablicy – postać ogólna:

```
array.forEach( function( element, index ) {
        // Instrukcje
});
```

- Parametr `element` – pozwala przekazać do funkcji kolejne elementy tablicy
- Parametr `index` – pozwala przekazać do funkcji indeksy kolejnych elementów tablicy

```
let names = [ 'Krystian',  'Monika',  'Danuta' ];

names.forEach( function( element, index ) {
    console.log( 'Element z Indexem: ' + index + ' ma wartość ' + element );
});
```

PĘTLE (II-***)

`Pętla for in` - pętla pozwala wykonywać iteracje na obiektach – postać ogólna:

```
for (let property in object) {
        // Instrukcje
}
```

- `object` – obiekt, po którym iterujemy
- `property` – zwracana do wnętrza pętli nazwa klucza

```
let person = { name: 'Krystian',  age: 35 };

for (let key in person) {
        console.log(person[key]); // Wypisze kolejno Krystian oraz 35
}
```

PĘTLE (II-***)

`Pętla while` służy, podobnie jak `for`, do wykonywania powtarzających się czynności.

Pętlę `for` najczęściej wykorzystuje się, kiedy liczba powtarzanych operacji jest znana, natomiast pętlę `while`, kiedy liczby powtórzeń nie znamy, a zakończenie pętli jest uzależnione od spełnienia pewnego warunku.

Ogólna postać pętli `while`:

```
while ( warunek ) {
        // instrukcje
}
```

Fragment kodu będzie powtarzany dopóki będzie spełniony warunek testowany w nawiasach.

```
let number = 0;
while(number < 2) {
    console.log(number++);  // wyloguje kolejno 0 oraz 1
}
```

PĘTLE (II-***)

Pętlą podobną do pętli `while` jest `pętla do...while`. Zasadniczą różnicą między tymi pętlami jest to, że w pętli `do...while` kod, który ma być powtarzany zostanie wykonany przed sprawdzeniem wyrażenia.

Wynika z tego, że instrukcje z wnętrza pętli `do...while` są wykonywane zawsze przynajmniej jeden raz, nawet jeśli warunek będzie fałszywy.

Ogólna postać pętli `do...while`:

```
do {
        // instrukcje
} while( warunek );
```

```
let number = 0;
do {
    console.log(number++);  // wyloguje kolejno 0 pomimo, że warunek jest nieprawdziwy
} while(number > 2);
```

PĘTLE (II-***)

Działanie każdej z pętli może być przerwane w dowolnym momencie za pomocą instrukcji `break`.

Jeśli zatem `break` pojawi się wewnątrz pętli, zakończy ona swoje działanie.

```
let i = 0;
while( true ) {

    /* pętla while wykonywała by się w nieskończoność (ponieważ warunek tej pętli był by
    zawsze prawdziwy), gdyby nie znajdująca się wewnątrz instrukcja break (dzięki czemu
    pętla będzie wykonywana dopóki wartość zmiennej i nie osiągnie co najmniej wartości 9) */

    console.log ( 'napis [i = ' + i + '] ' );

    if ( i++ >= 9 ) { break };
}
```

PĘTLE (II-***)

Instrukcja `continue` powoduje przejście do jej kolejnej iteracji.

Jeśli zatem wewnątrz pętli znajdzie się instrukcja `continue`, bieżąca iteracja (przebieg) zostanie przerwana oraz rozpocznie się kolejna (chyba że bieżąca iteracja była ostatnią).

```
for( let i = 1; i <= 20; i++ ) {

    if ( i % 2 != 0 ) { continue };
    /* jeśli wartość zmiennej i nie jest podzielna przez dwa to przejdź do kolejnej iteracji
    jeśli jest podzielna przez dwa to wypisz tą iterację */
    console.log ( i + ' ' );
}
```

Jest to pętla `for`, która wyświetla liczby całkowite z zakresu 1 – 20 podzielne przez 2.

JavaScript Object Notation - JSON (12-***)

JSON - Jest formatem do przechowywania i wymiany danych.

Jest używany gdy dane są przesyłane z serwera np. na stronę internetową.

JSON jest formatem tekstowym, bazującym na podzbiorze języka JavaScript.

Pomimo nazwy JSON jest formatem niezależnym od konkretnego języka. Wiele języków programowania obsługuje ten format danych przez dodatkowe pakiety bądź biblioteki.

JavaScript Object Notation - JSON (12-***)

```
{
    'employees': [
                {'firstName': 'John', 'lastName': 'Doe'},
                {'firstName': 'Anna', 'lastName': 'Smith'},
                {'firstName': 'Peter', 'lastName': 'Jones'}
            ]
}
```

Format JSON jest składniowo identyczny z kodem do tworzenia obiektów JavaScript:

- dane to pary nazwa-wartość
- dane są oddzielone przecinkami
- w klamrach zawarty jest obiekt
- w nawiasach kwadratowych jest tablica obiektów mających te same właściwości

WARSZTATY – KONTO W SERWISIE repl.it

Stwórz konto w serwisie
https://repl.it ☺

Jeśli go nie masz ;)

WARSZTATY – FUNKCJA ILOCZYN

Napisz funkcję, która pobiera trzy parametry.

Funkcja tworzy zmienną lokalną i do niej przypisuje iloczyn trzech pobranych parametrów.

Następnie funkcja zwraca wartość.

Zwórcona wartość funkcji jest przypisana do zmiennej globalnej, a potem wartość tej zmiennej jest wyślwietlana w konsoli.

Zadanie robimy z wykorzystaniem serwisu https://repl.it

Akademia 108

https://akademia108.pl