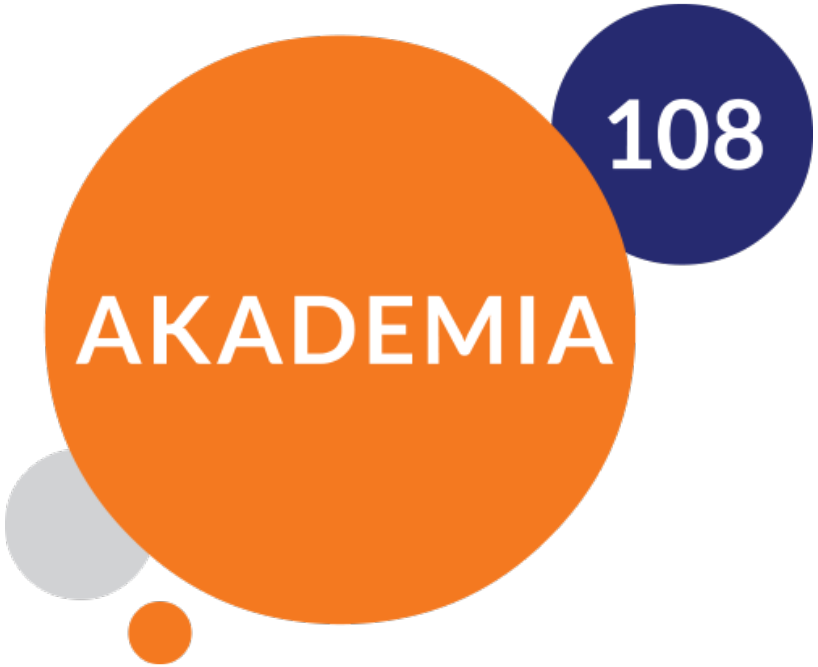


Kurs Front-End Developer

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
Cmd + Alt + / (Mac) CTRL + Shift + / (Windows & 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 + /

KOMENTARZE (I-***)

KOMENTARZ BLOKOWY

```
/*

... treść komentarza ...

*/
```

KOMENTARZ LINIOWY

```
// treść komentarza ...
```

KOMENTARZ LINIOWY W BLOKOWYM

```
/*

// treść komentarza

*/
```

ZMIENNE (2-***)

Zmienne to coś w rodzaju **„komórek pamięci”**, w których można przechowywać dane.

Zmienna posiada **nazwę**, dzięki której można się do niej odwołać w kodzie skryptu oraz **typ**, który określa jakie dane może przechowywać.

Zmienne tworzone są za pomocą **słowa kluczowego *var***, po którym następuje nazwa zmiennej, np.

var nazwaZmiennej;

Zmiennej można przypisać **wartość** za pomocą **operatora przypisania** czyli znaku = (równa się), co schematycznie można zapisać w ten sposób:

var nazwaZmiennej = wartoscZmiennej;

var liczba = 10;

lub po prostu:

nazwaZmiennej = wartoscZmiennej;

liczba = 10;

ZMIENNE (2-***)

Nazewnictwo zmiennych! - 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 *var*).

Warto nazywać zmienne tak aby wiadomo było do czego się odnoszą.

Należy też pamiętać o tym, że w JS istnieje rozróżnienie pomiędzy dużymi i małymi literami tzn.

var liczba; *oraz* ~~*var Liczba;*~~ - 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

FUNKCJE (3-***)

Funkcję możemy stworzyć także za pomocą **wyrażenia**. Jest to tak zwana **anonimowa funkcja** (czyli taką, która nie ma nazwy), którą od razu podstawiamy pod zmienna:

```
var nazwaFunkcji = function() {

    // kod funkcji anonimowej
}
nazwaFunkcji();     // wywołanie funkcji
```

FUNKCJE (3-***)

Funkcją można przekazywać **parametry (argumenty)**, 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-***)

Funkcja po zakończeniu działania zwraca jakąś wartość.

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.

Użycie samej instrukcji return, bez żadnych argumentów, powoduje przerwanie działania funkcji, w miejsce jej wywołania nie jest wtedy jednak podstawiana żadna wartość.

ZASIĘG ZMIENNYCH (4-***)

Gdy pracujemy z funkcjami 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.

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 DANYCH (5-***)

Typy referencyjne służą do zapisywania złożonych obiektów. Czyli wszystkie zmienne, które nie mają typu prostego, są typem referencyjnym np. obiekty (typ obiektowy), tablice.

Wartością zmiennych, które są typu referencyjnego jest adres wskazujący na miejsce, w pamięci, w którym znajdują się dane obiektu - zmienne nie mają przypisanej bezpośrednio wartości, a tylko wskazują na miejsce w pamięci, gdzie te dane są przetrzymywane.

TYPY DANYCH (5-***)

typ liczbowy (number)

Typ ten służy do reprezentacji liczbowej, np.

```
var liczba = 10;
```

Możliwe formaty zapisu :

- zapis liczb całkowitych i ułamkowych, np. 0, 1, -2, 3.0, 3.14, -6.28. 0pcjonalnie 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.

```
var zdanie = "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*)

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.

```
var varBol = true;
```

TYPY DANYCH (5-***)

typy specjalne (*null* i *undefined*)

Typ *undefined* oznacza po prostu typ niezdefiniowany. Jest on używany zarówno do oznaczenia braku wartości jak i wartości niezdefiniowanej.

null, podobnie jak w innych językach programowania, oznacza nic. W zasadzie może przypominać przeznaczeniem *undefined*, ale *null* został pomyślany raczej jako wyznacznik braku referencji do obiektu. W praktyce, z *null* spotkamy się używając funkcji wyszukujących element w dokumencie, np.

```
var element = document.getElementById( "id-elementu" );

if ( element !== null ) {

    // logika programu
}
```

TABLICE (6-***)

Tablice są to struktury danych pozwalające na przechowywanie uporządkowanego zbioru elementów.

Po utworzeniu tablicy za pomocą jednej z wcześniej podanych konstrukcji jest ona wypełniona wskazanymi wartościami, tzn. każda kolejna komórka zawiera kolejno podaną wartość.

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

nazwaTablicy[indeksKomorki]; np. *kolory[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** po prostu ustawiamy nową wartość w odpowiednim indeksie tablicy lub korzystamy z **metody *push()***, która dodaj nową wartość na końcu tablicy i zwraca jej długość:

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

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

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

TABLICE (6-***)

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

```
var imiona = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę
imiona.pop();                                        // usuwa ostatni element i zwraca jego wartość
console.log( imiona );                               // wypisze się "Marcin,Ania"
```

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

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

TABLICE (6-***)

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

```
var imiona = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę
imiona.shift();                                      // usuwa pierwszy element i go zwróci
console.log( imiona );                               // wypisze się "Ania,Agnieszka"
```

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

```
var imiona = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę
console.log( imiona. 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.

```
var imiona = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę

console.log( imiona.join() );          // wypisze się "Marcin,Ania,Agnieszka"
console.log( imiona.join( " - " ) );   // wypisze się "Marcin - Ania - Agnieszka"
console.log( imiona.join( " + " ) );   // wypisze się "Marcin + Ania + Agnieszka"
```

TABLICE (6-***)

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

```
var imiona = [ 'Marcin', 'Ania', 'Agnieszka' ];      //stwórz tablicę

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

Metoda *sort()* służy do sortowania tablicy.

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

OBIEKTY (7-***)

Każda wartość w tablicy ma swój index (klucz, numer porządkowy), dzięki któremu możesz się do niej odnieść.

Obiekt w JavaScript jest czymś „podobnym” do tablicy. Jest to także referencyjny typ danych.

Różnica polega na tym, że to my tworzymy klucze. Nie jesteśmy ograniczeni wyłącznie do kluczy numerycznych.

OBIEKTY (7-***)

```
var osoba = {
    name:    "Marcin",                              // właściwość obiektu
    height:  184,
    print :  function() { console.log( this.name ); }    // metoda obiektu
}
```

Wnioski, z powyższej konstrukcji:

- zmienna, która przechowuje obiekt, nazywa się instancją/obiektem,
- zamiast nawiasów [ ], przy pomocy których tworzysz tablicę, użyto nawiasów { },
- elementy składowe obiektu (pola) rozdzielone są przecinkiem,
- pary klucz-wartość są rozdzielone dwukropkiem klucz: wartość – są to właściwości obiektu. Programista sam decyduje jak nazwać klucz i jaką wartość może on przyjąć.
- obiekt może posiadać metody, są to działania które mogą być wykonywane na obiektach. Metody są to wewnętrzne funkcje przechowywane we właściwościach obiektów.

OBIEKTY (7-***)

Dostęp do właściwości obiektu:

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

np. w odniesieniu do przykładu z poprzedniego slajdu

osoba.name; lub *osoba["name"];*

Dostęp do metod obiektu:

nazwaObiektu.nazwaMetody();

np.

osoba.print();

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

OBIEKTY (7-***)

Dodawanie właściwości:

```
var osoba = {
    name:  "Marcin",                                    // właściwość obiektu
    height:  184,
    print :  function() { console.log( this.name );}    // metoda obiektu
}

osoba.weight = 73;                                      // dodawanie własności
osoba.printDetail = function() {                        // dodawanie metody
        return this.name + " " + this.height + " " + this.weight;
}
```

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.

W wielu językach programowania klasę definiujemy za pomocą słowa kluczowego ***class***. JavaScript pozwala to zrobić na dwa sposoby. Do stworzenia klasy obiektu wykorzystujemy słowo kluczowe **function** lub słowo kluczowe **class**. W słowa class potrzebny jest także **constructor()**.

Pojedynczy obiekt stworzony na podstawie klasy, to instancja klasy.

KLASY (7-***)

```
//Tworzymy klasę obiektu Osoba
class Osoba {
        constructor(imie, nazwisko) {
                this.imie = imie;
                this.nazwisko = nazwisko;
        }
        wyswietlInfo() {
                console.log( "Imię: " + this.imie + ",  " + "Nazwisko: " + this.nazwisko);
        }
}

var krystian = new Osoba('Krystian', 'Dziopa');  // stwórz nową instancję obiektu Osoba

krystian.wyswietlInfo();  //Wypisze „Imię: Krystian, Nazwisko: Dziopa

var lukasz = new Osoba('Łukasz', 'Badocha');  // stwórz nową instancję obiektu Osoba

lukasz.wyswietlInfo();  //Wypisze „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.

OBIEKT MATH (8-***)

Stałe matematyczne dostępne dzięki obiektowi Math (własnością):

*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(ilosc-stopni) - zwraca cosinus
Math.pow(podstawa , wykladnik) - zwraca liczbę podniesioną do potęgi
Math.rondom( ) - zwraca przypadkową 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.

Operatory można podzielić na:

- arytmetyczne
- porównania
- przypisania
- logiczne
- warunkowe

OPERATORY – OPERATORY ARYTMETYCZNE (9-***)

OPERATOR	WYKONYWANE DZIAŁANIE
*	mnożenie
/	dzielenie
+	dodawanie
-	odejmowanie
%	dzielenie modulo (reszta z dzielenia)
++	inkrementacja (zwiększanie)
--	dekrementacja (zmniejszanie)

OPERATORY – OPERATORY PRZYPISANIA (9-***)

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

OPERATOR	OPIS
=	przypisanie wartości
+=	przypisanie argumentowi umieszczonemu z lewej strony wartość wynikającej z dodawania argumentu znajdującego się z lewej strony i argumentu znajdującego się z prawej strony operatora
-=	przypisanie argumentowi umieszczonemu z lewej strony wartość wynikającej z odejmowania argumentu znajdującego się z lewej strony i argumentu znajdującego się z prawej strony operatora
*=	przypisanie argumentowi umieszczonemu z lewej strony wartość wynikającej z pomnożenia argumentu znajdującego się z lewej strony i argumentu znajdującego się z prawej strony operatora
/=	przypisanie argumentowi umieszczonemu z lewej strony wartość wynikającej z podzielenia argumentu znajdującego się z lewej strony i argumentu znajdującego się z prawej strony operatora
%=	przypisanie argumentowi umieszczonemu z lewej strony wartość wynikającej z dzielenia modulo argumentu znajdującego się z lewej strony i argumentu znajdującego się z prawej strony operatora

OPERATORY – OPERATORY PORÓWNANIA (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.

OPERATOR	OPIS
==	równe
!=	różne
===	równa wartość i taki sam typ danych
!==	różne i inny typ danych
>	większe od
<	mniejsze od
>=	większe bądź równe od
<=	mniejsze bądź równe od

OPERATORY – OPERATORY LOGICZNE (9-***)

Operatory logiczne - za pomocą operatorów logicznych możemy łączyć kilka porównań w jedną całość.

Można je wykonywać na argumentach, które posiadają wartość logiczną: prawda lub fałsz.

Wynikiem takiej operacji jest wartość prawda lub fałsz.

Operatory logiczne:

- iloczyn logiczny (AND) - &&
- suma logiczna (OR) - ||
- negacja logiczna (NOT) - !

OPERATORY – OPERATORY LOGICZNE (9-***)

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.

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.

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.

OPERATORY – OPERATOR WARUNKOWY (9-***)

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

warunek ? wyrazenie1 : wyrazenie2

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

```
var liczba = 100;
var wynik = ( liczba < 0 ) ? -1 : 1;
console.log( wynik );      // wynik = 1
```

INSTRUKCJE WARUNKOWE (10-***)

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

Instrukcje warunkowe mogą być zagnieżdżane.

Instrukcje warunkowe:

- *if*
- *if-else*
- *else if*
- *switch*

INSTRUKCJE WARUNKOWE – IF (10-***)

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.

```
var x = 1;
if ( x == 1 ) {

    console.log( 'Liczba równa się 1' );
}
```

INSTRUKCJE WARUNKOWE – IF-ELSE (10-***)

Poprzez dodanie do instrukcji *if* bloku *else* możemy sprawdzić przeciwieństwo warunku *if* – **instrukcja *if-else***:

```
if ( warunek ) {

        // instrukcje do wykonania jeśli warunek jest spełniony
} else {

        // instrukcje do wykonania jeśli warunek nie jest spełniony
}

var liczba = -1;
if ( liczba < 0 ) {

        console.log ( "Wartość zmiennej liczba jest mniejsza od 0." );
} else {

        console.log ( "Wartość zmiennej liczba nie jest mniejsza od 0." );
}
```

INSTRUKCJE WARUNKOWE –ELSE IF (10-***)

Trzecia wersja instrukcji *if* pozwala na badanie wielu warunków. Po bloku *if* może wystąpić wiele dodatkowych bloków *else if* – ***instrukcja 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 –SWITCH (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
}
```

INSTRUKCJE WARUNKOWE –SWITCH (10-***)

Wcześniejszy zapis należy rozumieć następująco:

- sprawdź wartość wyrażenia **wyrażenie**, jeśli wynikiem jest **wartość1**, to wykonaj **intrukcje1** i przerwij wykonywanie bloku *switch* (przerwanie jest wykonywane przez **instrukcję *break***); jeśli wynikiem jest **wartość2**, to wykonaj **intrukcje2** itd.

- jeśli nie zachodzi żaden z wymienionych przypadków, wykonaj **instrukcje4** i zakończ blok *switch*

- blok *default* jest jednak opcjonalny i może zostać pominięty.

PĘTLE (II-***)

Pętle w programowaniu pozwalają nam wykonywać dany kod pewną ilość razy.

Pętle możemy zagnieżdżać.

Pętle występujące w języku JavaScript możemy podzielić na dwa główne rodzaje:

- pętle typu for (w tym *for*)
- pętle typu while (w tym *while* i *do...while*)

PĘTLE – FOR (II-***)

Ogólna postać **pętli *for***

```
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

PĘTLE – FOREACH (11-***)

Ogólna postać **pętli *forEach***

```
var tablica = [ ”Krystian”,  ”Monika”,  ”Danuta” ];

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

element jest wartością elementu tablicy

index jest indexem elementu tablicy

PĘTLE – WHILE (11-***)

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.

PĘTLE – DO…WHILE (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 );
```

PĘTLE – PRZERYWANIE PĘTLI (11-***)

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.

```
var 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 – KONTYNUACJA PĘTLI (11-***)

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( var 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

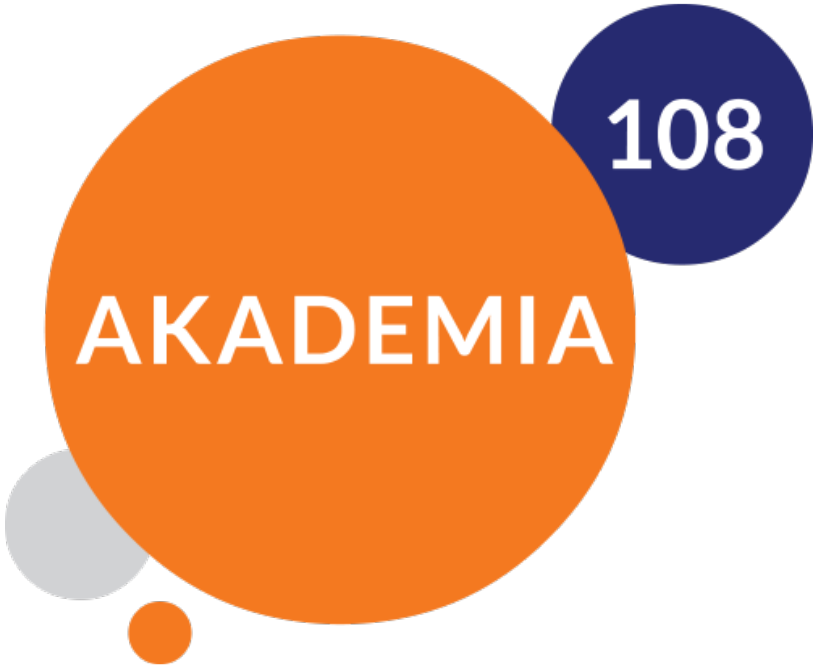
Napisz funkcję, która pobiera trzy parametry.

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