Kurs Front-End Developer

JavaScript

KOMENTARZE

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/CTRL + Alt + /

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

KOMENTARZ BLOKOWY

```
/*

... treść komentarza ...

*/
```

KOMENTARZ LINIOWY

```
// treść komentarza ...
```

KOMENTARZ LINIOWY W BLOKOWYM

```
/*

// treść komentarza

*/
```

ZMIENNE

Zmienne to coś w rodzaju „pudełek", 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

Zasięg zmiennej i deklaracja ze słowem var

Jeśli deklarację zmiennej wewnątrz funkcji poprzedza słowo *var* zmienna ta nie jest dostępna dla innych funkcji (dlatego kilka funkcji może deklarować i używać zmiennej o tej samej nazwie bez żadnej kolizji oznaczeń). Jeśli przed pierwszym przypisaniem wartości zmiennej wewnątrz funkcji nie ma słowa *var* zostanie utworzona lub zmodyfikowana zmienna globalna i inne funkcje będą mogły odczytać jej wartość.

ZMIENNE

Przy tworzeniu nazw zmiennych stosuje się następujące zasady:

- 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*, *alert* 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

TYPY DANYCH

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

Wartością zmiennych, które są typu prostego jest faktyczna wartość zmiennej – zmienne mają przypisaną bezpośrednio wartość.

TYPY DANYCH

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

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

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:

- \b - backspace (ang. *backspace*)
- \n - nowy wiersz (ang. *new line*)
- \r - powrót karetki (ang. *carriage return*)
- \f - nowa strona (ang. *form feed*)
- \t - tabulacja pozioma (ang. *horizontal tab*)
- \" - cudzysłów (ang. *double quote*)
- \' - apostrof (ang. *single quote*)
- \\ - lewy ukośnik (ang. *backslash*)

TYPY DANYCH

Metody dla zmiennych string

- **charAt** - znak na danej pozycji
- **concat** - łączenie dwóch ciągów (równoznaczne z +=)
- **indexOf** - pozycja szukanego ciągu znaków
- **lastIndexOf** - ostatnia pozycja szukanego ciągu znaków
- **replace** - zamiana jednego ciągu znaków na drugi
- **slice** - wyciągnięcie kawałka danego ciągu
- **split** - dzielenie ciągu na podstawie danego rozdzielnika
- **substr** - wyciągnięcie kawałka danego ciągu
- **substring** - wyciągnięcie kawałka danego ciągu
- **toLowerCase** - zamiana wszystkich znaków na małe
- **toUpperCase** - zamiana wszystkich znaków na wielkie
- **trim** - usunięcie wszystkich białych znaków z początka i końca

TYPY DANYCH

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

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
}
```

TYPY DANYCH

typ tablicowy (*Array*)

Tak jak obiekty, tablice są typem złożonym, Służą one do grupowania danych w strukturę, gdzie każdemu elementowi przypisany jest określony indeks.

Tablice w JavaScript są obsługiwane przez klasę *Array*. Obiekt tej klasy można utworzyć na trzy sposoby, poprzez wywołanie odpowiedniego konstruktora:

- utworzenie pustej tablicy: `var tablica = new Array();`
- utworzenie tablicy i podanie jej rozmiaru (np. 10 elementów):
  `var tablica = new Array( 10 );`
- utworzenie tablicy i podanie listy elementów które mają się w niej znaleźć:
  `var tablica = new Array( 1, 2, 3, 4, 5 );`

TYPY DANYCH

Istnieje też możliwość utworzenia tablicy poprzez umieszczenie kolejnych jej elementów jako listy w nawiasach kwadratowych.

- utworzenie pustej tablicy: `var tablica = [ ];`
- utworzenie tablicy zawierającej jeden element o wartości 10:
  `var tablica = [ 10 ];`
- utworzenie tablicy zawierającej więcej elementów:
  `var tablica = [ 1, 2, 3, 4, 5 ];`

Każdy element w tablicy posiada przypisaną pozycję (indeks w tablicy). Numeracja indeksów zaczyna się od zera. Do każdego elementu w tablicy można się odwołać podając w nawiasach kwadratowych indeks elementu.

TYPY DANYCH

typ obiektowy (*Object*)

Służy do reprezentacji obiektów. Najczęściej wykorzystuje się obiekty wbudowane oraz udostępniane przez przeglądarkę.

Jest to także typ złożony, podobnie jak tablice. Oznacza to że wewnątrz nich można przechowywać więcej niż jedną wartość. Posiada on jednak zupełnie inne możliwości jak i zastosowania - można wewnątrz niego zdefiniować funkcje (zwane także metodami) oraz pola (zwane także właściwościami, zmiennymi). Zarówno do jednych i drugich można się odwołać podobnie jak do "zwykłych" zmiennych i funkcji, trzeba tylko przed ich nazwą umieścić obiekt którego są elementami (czyli np. zmienną która przechowuje dany obiekt) i kropkę.

W połączeniu z obiektami często występuje także pojęcie "klasy". Klasa jest do definicja danego obiektu (czyli informacja co on ma robić, itd.), natomiast obiekt jest to konkretny egzemplarz klasy.

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

METODY MATH

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.

METODY MATH

Stałe matematyczne dostępne dzięki obiektowi Math (własnością):

E	-	zwraca stałą Eulera, która wynosi ok. 2.71
LN2	-	zwraca logarytm z dwóch, tj. ok. 0.69
LN10	-	zwraca logarytm z dziesięciu, tj. ok. 2.30
LOG2E	-	zwraca logarytm o podstawie 2 z liczby E, czyli ok. 1.44
LOG10E	-	zwraca logarytm o podstawie 10 z E, czyli ok. 0.43
PI	-	zwraca wartość liczby Pi, czyli ok. 3.14
SQRT1_2	-	zwraca pierwiastek kwadratowy z 0.5, czyli ok. 0.70
SQRT2	-	zwraca pierwiastek kwadratowy z 2, czyli ok. 1.41

METODY MATH

```
document.write( "E = " + Math.E + "<br />" );

document.write( "LN2 = " + Math.LN2 + "<br />" );

document.write( "LN10 = " + Math.LN10 + "<br />" );

document.write( "LOG2E = " + Math.LOG2E + "<br />" );

document.write( "LOG10E = " + Math.LOG10E + "<br />" );

document.write( "PI = " + Math.PI + "<br />" );

document.write( "SQRT1_2 = " + Math.SQRT1_2 + "<br />" );

document.write( "SQRT2 = " + Math.SQRT2 + "<br /><br />" );
```

METODY MATH

Operacje matematyczne dostępne dzięki metodom Math:

abs(liczba)	-	Zwraca wartość absolutną liczby
acos(liczba)	-	Zwraca arcus cosinus z liczby (podanej w radianach)
asin(liczba)	-	Zwraca arcus sinus z liczby (podanej w radianach)
atan(liczba)	-	Zwraca arcus tangens z liczby (podanej w radianach)
ceil(liczba)	-	Zwraca najmniejszą liczbę całkowitą, większą lub równą podanej liczbie
cos(liczba)	-	Zwraca cosinus liczby (podanej w radianach)
exp(liczba)	-	Zwraca wartość E podniesionej do potęgi wyrażonej podanym argumentem

METODY MATH

floor(liczba)	-	Zwraca największą liczbę całkowitą mniejszą lub równą podanej liczbie
log(liczba)	-	Zwraca logarytm naturalny liczby
max(liczba1, liczba2)	-	Zwraca większą z dwóch liczb
min(liczba1, liczba2)	-	Zwraca mniejszą z dwóch liczb
pow(liczba1, liczba2)	-	Zwraca wartość liczby1 podniesionej do potęgi liczby2
random()	-	Zwraca wartość pseudolosową z przedziału 0 - 1
round(liczba)	-	Zwraca zaokrąglenie danej liczby do najbliższej liczby całkowitej
sin(liczba)	-	Zwraca sinus liczby (podanej w radianach)
sqrt(liczba)	-	Zwraca pierwiastek kwadratowy liczby
tan(liczba)	-	Zwraca tangens liczby (podanej w radianach)

METODY MATH

```
var zmienna1 = 56.5;
var zmienna2 = 74.3;

document.write( Math.min( zmienna1, zmienna2 ) ); //wypisze sie 56.5
document.write( Math.cos( 0 ) );                  //wypisze sie 1
document.write( Math.round( zmienna1 ) );         // da w rezultacie 57
document.write( Math.ceil( zmienna1 ) );          // da w rezultacie 57
```

INSTRUKCJE WARUNKOWE

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 – INSTRUKCJA IF

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 – INSTRUKCJA IF-ELSE

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 ) {

        document.write( "Wartość zmiennej liczba jest mniejsza od 0." );
} else {

        document.write( "Wartość zmiennej liczba nie jest mniejsza od 0." );
}
```

INSTRUKCJE WARUNKOWE – INSTRUKCJA ELSE IF

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 – INSTRUKCJA ELSE IF

```
var x = 5;

if ( x > 5 ) {

    console.log( 'Liczba jest większa od 5' );

} else if ( x < 5 ) {

    console.log( 'Liczba jest mniejsza od 5' );

} else {

    console.log( 'Liczba równa się 5' );
}
```

INSTRUKCJE WARUNKOWE – INSTRUKCJA SWITCH

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 – INSTRUKCJA SWITCH

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.

INSTRUKCJE WARUNKOWE – INSTRUKCJA SWITCH

```
var numer = 4;
// poniższy warunek zwróci "Numer równa się cztery"
switch ( numer ) {
    case 1:
        console.log( 'Numer równa się jeden' );
        break;
    case 2:
        console.log( 'Numer równa się dwa' );
        break;
    case 3:
        console.log( 'Numer równa się trzy' );
        break;
    case 4:
        console.log( 'Numer równa się cztery' );
        break;
    default:
        console.log( 'Nie wiem ile równa się numer' );
}
```

PĘTLE

Pętlę 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 – PĘTLA FOR

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 – PĘTLA FOR

```
for( var liczba = 0; liczba < 10; liczba++ ) {

  document.write( liczba + " " );
}
```

Należy to rozumieć następująco:

- utwórz zmienną liczba i przypisz jej wartość zero (liczba = 0),
- dopóki wartość zmiennej liczba jest mniejsza od 10 (liczba < 10), wykonuj instrukcje znajdujące się wewnątrz pętli,
- oraz zwiększaj zmienną liczba o jeden (liczba++).

Tym samym na ekranie pojawi się ciąg liczb od 0 do 9 odzwierciedlających kolejne stany zmiennej liczba, którą nazywamy zmienną iteracyjną, czyli kontrolującą kolejne przebiegi (iteracje) pętli.

PĘTLE – PĘTLA WHILE

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 – PĘTLA WHILE

```
var i = 0;

while( i < 10 ) {

  document.write( "Pętla while [i = " + i + "]" );
  document.write( "<br />" );

  i++;
}
```

Pętla będzie wykonywana dopóki wartość zmiennej i będzie mniejsze od 10.

PĘTLE – PĘTLA DO…WHILE

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 – PĘTLA DO...WHILE

```
var i = 0;

do {

    document.write( "Pętla do...while [i = " + i + "]" );
    document.write( "<br />" );
} while( i++ < 9 );
```

Pętla wykona się 10 razy.

PĘTLE – PRZERYWANIE PĘTLI

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) */

    document.write( "napis [i = " + i + "] <br />" );
    if ( i++ >= 9 ) break;
}
```

PĘTLE – KONTYNUACJA PĘTLI

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 */
    document.write( i + " " );
}
```

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

OPERATORY

Na zmiennych można wykonywać róże operacje za pomocą operatorów.

Operatory można podzielić na:

- arytmetyczne
- porównania
- przypisania
- logiczne
- warunkowe

OPERATORY – OPERATORY ARYTMETYCZNE

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

OPERATORY – OPERATORY ARYTMETYCZNE

```
x = 5;
y = x + 2;      // y = 7
y = x - 1;      // y = 4
y = x * 3;// y = 15
y = x / 2;      // y = 2.5
y = x % 2;      // 1 bo % oznacza resztę z dzielenia
x--;            // to to samo co x = x-1
x++;            // to to samo co x = x+1
y = x--         // y = 4, x = 5
y = --x         // y = 5, x = 4
```

OPERATORY – OPERATORY PRZYPISANIA

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 PRZYPISANIA

```
y = 5;          // y = 5
y += 2;         // y = 7
y -= 1;         // y = 4
y *= 3;         // y = 15
y /= 2;         // y = 2.5
y %= 2;         // 1 bo % oznacza resztę z dzielenia
```

OPERATORY – OPERATORY PORÓWNANIA

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 PORÓWNANIA

```
var myVar = 8;

if ( myVar == 10 ) {

    // ten kawałek kodu się nie wykona
}
```

OPERATORY – OPERATORY LOGICZNE

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

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 – OPERATORY LOGICZNE

```
var myVar = 8;
var myVar2 = 15;

if ( myVar == 8 && myVar2 == 10 ) {
        // ten kawałek kodu się nie wykona bo mamy "i"
}

if ( myVar == 8 || myVar2 == 8 ) {
        // ten kawałek się wykona bo mamy "lub"
}

if ( !( myVar == 8 ) ) {
        // ten kawałek się nie wykona, bo mamy negację!
        // powyższy warunek jest jednoznaczny z myVar != 8
}
```

OPERATORY – OPERATOR WARUNKOWY

Operator warunkowy 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;
document.write( wynik );      // wynik = 1
```

FUNKCJE

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

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

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

Sposoby tworzenia funkcji różnią się od siebie nie tylko zapisem, ale także tym jak są one interpretowane przez przeglądarkę.

Funkcja za pomocą deklaracji jest od razu dostępna dla całego skryptu.

Funkcja stworzona za pomocą wyrażenia jest dostępna dopiero po całkowitym przetworzeniu skryptu.

FUNKCJE

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

```
function dodaj( wart1, wart2 ) {       //deklaracja funkcji o nazwie dodaj

    // instrukcje wewnątrz funkcji
    var suma = wart1 + wart2;      // zmienna przechowująca sumę argumentów funkcji

    // wypisz w dokumencie
    document.write( "Wynikiem dodawania " + wart1 + " oraz " + wart2 + " jest " );
    document.write( suma );
    document.write( "." );
}
dodaj( 11, 12 );        // wywołaj funkcję dla argumentów wart1=11 oraz wart2=12
document.write( "<br/>" );
dodaj( 5, 8 );          // wywołaj funkcję dla argumentów wart1=5 oraz wart2=8
```

FUNKCJE

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.

Wywołanie funkcji z return, w miejscu jej wywołania wstawi zwracaną przez funkcję 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

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.

ZASIĘG ZMIENNYCH

Zmienne globalne tworzymy definiując je za pomocą słowa kluczowego *var* po za funkcją.

```
var wart1 = 4;                          // zmienna globalna
var wart2 = 5;                          // zmienna globalna

function dodaj() {          //deklaracja funkcji o nazwie dodaj

        // instrukcje wewnątrz funkcji
        // odwołanie do zmiennych globalnych
        document.write( "Wynikiem dodawania " + wart1 + " oraz " + wart2 + " jest " );
        document.write( wart1 + wart2 );
        document.write( "." );
}

dodaj();
```

ZASIĘG ZMIENNYCH

Natomiast zmienne lokalne tworzymy definiując je za pomocą słowa kluczowego *var* wewnątrz funkcji. Taka zmienna dostępna jest tylko wewnątrz funkcji w której została zdefiniowana i nie możemy się do nie odwołać po za tą funkcją.

```
function dodaj() {          //deklaracja funkcji o nazwie dodaj

        // instrukcje wewnątrz funkcji
        var wart1 = 4;                          // zmienna lokalne
        var wart2 = 5;                          // zmienna lokalne

        var suma = wart1 + wart2;               // zmienna suma jest zmienną lokalną

        // wypisz w dokumencie – odwołanie się do zmiennych lokalnych
        document.write( "Wynikiem dodawania " + wart1 + " oraz " + wart2 + " jest " );
        document.write( suma );
        document.write( "." );
}

dodaj();
```

ZASIĘG ZMIENNYCH

Aby stworzyć zmienną lokalną konieczna jest jej deklaracja ze słowem kluczowym *var* wewnątrz funkcji.

To oznacza, że jeżeli w funkcji w pierwszym odwołaniu do jakiejś zmiennej nie użyjemy słowa *var*, to zostanie to potraktowane jak odwołanie do zmiennej globalnej. Co więcej, jeżeli taka zmienna jeszcze nie istnieje, to zostanie utworzona!

```
var wart1 = 4;                          // zmienna globalna

function dodaj() {                      //deklaracja funkcji o nazwie dodaj

    // instrukcje wewnątrz funkcji
    var wart2 = 5;                      // deklaracja zmienna lokalne
    var suma;

    suma = wart1 + wart2;               // przypisanie wartości zmiennej lokalnej

    return suma;
}
document.write( dodaj() );
```

TABLICE

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

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] );        // wypisze się "Piotrek i Grzegorz"

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

TABLICE

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

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

Można wykonywać również **pętle po tablicy**. Aby zrobić pętlę po tablicy należy skorzystać z jednej z dwóch pętli:

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

for ( var i=0; i < imiona.length; i++ ) {
    console.log( 'Imię numer ' + i + ': ' + imiona[ i ] );
}

// lub

imiona.forEach( function( el, i ) {
    console.log( 'Imię numer ' + i + ': ' + el );
} )
```

TABLICE

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

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"
```

TABLICE

Do łączenia tablic służy **metoda *concat(array1, ...)***, która jako parametr przyjmuje jedną lub kilka tablic.

```
var animals = [ "Pies", "Kot" ];
var animals2 = [ "Słoń", "Wieloryb" ];
var animals3 = [ "Chomik", "Świnka" ];

var newTableSmall = animals.concat( animals2 );
var newTableBig = animals.concat( animals2, animals3 );

console.log( newTableSmall );      // wypisze ["Pies", "Kot", "Słoń", "Wieloryb"]
console.log( newTableBig );        // wypisze ["Pies", "Kot", "Słoń", "Wieloryb", "Chomik", "Świnka"]
```

TABLICE

Pozostałe metody które mogą się przydać na początek pracy z tablicami:

tablica.indexOf("string")	-	Szuka pierwszego wystąpienia elementu o danej wartości i wskazuje jego pozycję w tablicy
Array.isArray(tablica)	-	Sprawdza czy obiekt jest tablicą
tablica.slice(1, 3) obu stron	-	Wybiera część tablicy i zwraca nową tablice – obcina z
tablica.splice(2, 0)	-	Dodaje/Usuwa elementy z tablicy
tablica.toString()	-	Przekształca tablicę na ciąg znaków i zwraca wynik

OBIEKTY

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.

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

OBIEKTY

```
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 przechowywane we właściwościach jak **definicje funkcji**.

OBIEKTY

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

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

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*. W języku JavaScript nie występuje słowo kluczowe *class*. Do stworzenia klasy obiektu wykorzystujemy funkcję czyli słowo kluczowe function.

Pojedynczy obiekt stworzony na podstawie klasy, to instancja klasy.

KLASY

```
//Tworzymy klasę obiektu Osoba
function Osoba(imie, nazwisko) {
    this.imie = imie;
    this.nazwisko = nazwisko;
    this.wyswietlInfo = function() {
        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
```

JavaScript Object Notation - JSON

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

```
{
"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

JavaScript Object Notation - JSON

Wartości w **JSON** mogą być:

- liczby (całkowite lub zmiennoprzecinkowe)
- ciągi znaków (w cudzysłowach)
- wartości logiczne (prawda lub fałsz)
- tablice (w nawiasach kwadratowych)
- obiekty (w nawiasach klamrowych)
- null

Obiekty i tablice mogą być dowolnie zagnieżdżane.

JavaScript Object Notation - JSON

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

Obiekt „employees” jest tablicą, która zawiera trzy obiekty. Każdy z tych trzech obiektów jest osobą posiadającą imię i nazwisko (właściwości obiektu).

JavaScript Object Notation - JSON

Aby użyć danych w formacie JSON w JavaScript należy:

- utworzyć tablicę obiektów i przypisać do niej dane

```
var employees = [
    {"firstName": "John", "lastName": "Doe"},
    {"firstName": "Anna", "lastName": "Smith"},
    {"firstName": "Peter", "lastName": "Jones"}
];
```

- pierwszy wpis z tablicy obiektów można uzyskać w następujący sposób (zwróci John Doe):

```
employees[0].firstName + " " + employees[0].lastName;
```

lub

```
employees[0]["firstName"] + " " + employees[0]["lastName"];
```

JavaScript Object Notation - JSON

Aby użyć danych w formacie JSON jako obiektu JavaScript należy:

- utworzyć ciąg JS zawierający dane JSON

```
var text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

- zamienić ciąg JS za pomocą **funkcji *JSON.parse()*** na obiekt JS

```
var obj = JSON.parse(text);
```

- gotowe można używać nowego obiekt na stronie

```
obj. employees[0].firstName + " " + obj. employees[0].lastName;  // zwróci John Doe
```

Akademia 108

ul. Mostowa 6/13

31-061 Kraków