Java Script

Cz.1

JavaScript jest językiem skryptowym przeznaczonym do wykorzystania na stronach internetowych.

Co to oznacza? Dzięki niemu możesz w łatwy sposób poprawić interaktywność na swoich stronach. Możesz w odpowiedni sposób oddziaływać na poczynania użytkownika. Możesz robić naprawdę dużo fajnych rzeczy ;). A dzięki temu, że wszystkie czynności wykonywane są po stronie klienta - wszystko to wykonuje się w błyskawicznym tempie.

Jak działa Javascript

Ogólnie rzecz ujmując w Javascript praktycznie wszystko jest jakimś obiektem.

Obiektami są więc tablice, funkcje, własne obiekty, daty czy nawet teksty lub liczby.

Wszystkie dane z jakich tutaj korzystamy możemy podzielić na dwa typy: typy proste i typy złożone (obiekty). Typy proste służą do przechowywania prostych typów danych. Liczby, teksty, wartości nieistniejące. Coś bardzo prostego - stąd ich nazwa. Typy obiektowe (referencyjne) służą do przechowywania bardziej złożonych danych - czyli obiektów.

Wymagania

Jeżeli chodzi o przeglądarki - wedle uznania. Jeżeli chcesz używać starych IE, nie wnikam w twoje zapędy.

Osobiście polecam którąś z nowych przeglądarek, szczególnie Chrome, który nie dość, że działa bardzo szybko, to ma rewelacyjny debuger.

Jeżeli chodzi o edytor tekstu, to Notepad++ w zupełności wystarczy.

Będąc w każdej porządnej przeglądarce naciskamy klawisz F12, co włącza nam debugera na dole okna.

Wstawiamy pierwszy skrypt na naszą stronę

Do umieszczenia skryptów na stronie służy znacznik <script>:

```
<html>
<head>
    <title>Super skrypt</title>
</head>
<body>
<script>
    ...instrukcje skryptu
</script>
</body>
```

UWAGA: W przypadku html4 trzeba do tagu script dodać dodatkowy atrybut type="javascript".

Umieszczanie skryptów w oddzielnym pliku

Ogólnie rzecz biorąc powinniśmy unikać wstawiania skryptów bezpośrednio w kod strony, zastępując je skryptami umieszczonymi w dołączanych plikach. Nie jest to zasada, której pominięcie zniszczy nam życie – od każdej zasady są odstępstwa – jednak dobrze jest się jej trzymać…

Umieszczanie skryptów w oddzielnym pliku

Aby przyłączyć plik ze skryptami do naszej strony powinniśmy zastosować dodatkowy atrybut src:

```
<script src="plik_ze_skryptem.js"></script>
```

Umieszczanie skryptów w oddzielnym pliku

Skrypty możesz wrzucać w różne miejsca na stronie. Najlepszymi miejscami do umieszczania skryptów są sekcja HEAD i koniec sekcji BODY:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Jestem super stroną</title>
    <script src="modernizr.js"></script>
    <script src="jquery.js"></script>
  </head>
  <body>
    ..treść strony...
    <script src="super-script.js"></script>
    <script>
      alert('Witaj świecie');
    </script>
  </body>
</html>
```

Wybór miejsca wiąże się z dwiema kwestiami.

Pierwsza z nich tyczy się zasady, która mówi o tym, by użytkownik dostawał w pierwszej chwili treść strony. Cała interakcja (JS), dodatki itp powinna być wysyłana w drugiej kolejności.

Ale to drobnostka, o której na początku możemy zapomnieć.

Ta "bardziej ważna" sprawa tyczy się samych skryptów. Bardzo często będziesz chciał w nich działać na konkretnych elementach strony. Przykładowo zapragniesz pokazać przycisk, ukryć formularz, pokazać użytkownikowi jakieś okienko itp. Jeżeli twój skrypt zostanie wczytany przed takim elementem (czyli w kodzie HTML będzie się znajdował przed takim elementem HTML), a nie posiada żadnych metod do wykrywania czy cała strona została już wczytana, to dostaniesz błąd wynikający z tego, że skrypt ten nie widzi elementu.

ZMIENNE

Zmienne to coś w rodzaju "pudełek" w których możemy przechowywać pewne dane:

```
<script>
    var someVar1 = 'Przykładowy tekst';
    var someVar2 = 230;
    var someVar3 = Number.MAX_VALUE;
</script>
```

Nazewnictwo zmiennych

Nazwy zmiennych które deklarujemy nie mogą być byle jakie. Istnieją pewne zasady których musimy się trzymać. I tak:

-każda nazwa zmiennej musi się zaczynać od litery (A-Z, a-z), lub znaku podkreślenia ("_"),

-nazwa zmiennej nie może się zaczynać od cyfry (0-9),

-nazwa zmiennej nie może zawierać spacji (można zamiast spacji używać podkreślenia),

-nazwa zmiennej nie może zawierać polskich liter,

-nazwą zmiennej nie może być słowo kluczowe zarezerwowane przez JavaScript.

Tak naprawdę nie musisz wkuwać powyższych punktów na pamięć. Po prostu nie zaczynaj nazw od cyfr. Tyle. Jest jednak ważniejsza sprawa, którą zapamiętaj.

Nazywaj swoje zmienne tak, by dało się zrozumieć do czego się odnoszą.

Typy danych

W Javascript dane dzielą się na 2 typy: typy proste oraz referencje.

Dane typu prostego reprezentują proste typy danych - liczby, teksty, wartości boolowskie (prawda/fałsz), niezidentyfikowane (undefined) oraz null.

```
//przykład danych typu prostego
var varNr = 20; //prosta liczba
var text = 'To jest tekst'; //prosty łańcuch znaków
var varBol = true; //logiczny - prawda/fałsz
var someVar = null; //nic

undefined //zmienna nie określona, nie istniejąca
```

Proste typy danych jak sama nazwa wskazuje są prostymi bytami, które poza wartością nic w sobie nie mają.

Żeby wykonywać na nich jakieś operacje (np. pobrać długość tekstu) musiałbyś albo za każdym razem używać na nich osobnych funkcji (tak jak w starszych PHP), albo musiałby by one być skonwertowane na obiekty, które dawały by dostęp do odpowiednich metod i właściwości.

Na szczęście podczas pracy JS poza sceną dokonuje automatycznej tymczasowej konwersji typu prostego na obiekt, odpala użytą właściwość lub metodę, a następnie przywraca daną zmienną do typu prostego:

```
var ourText = 'Przykładowy tekst'; //deklarujemy prostą zmienną

ourText.length  //js poza sceną skonwertowało ourText na obiekt String, sprawdziło jego długość
```

Dzięki powyższej konwersji w javascript wszystko zachowuje się jak obiekt. Zasada ta nie dotyczy się tylko undefined i null, które nie potrzebują mieć właściwości i metod.

Wszystkie zmienne nie będące typem prostym są obiektami i są typu referencyjnego (są obiektami). Ten typ danych charakteryzuje sie tym, że zmienne nie mają przypisanej bezpośrednio wartości, a tylko wskazują na miejsce w pamięci, gdzie te dane są przetrzymywane.

```
var arr1 = [1, 2, 3];
var arr2 = arr1; //zmienna arr2 wskazuje na tablicę [1, 2, 3]

arr1.push(4);
```

Ogólnie: typy proste są proste. To co nie jest typem prostym jest obiektem. Cała reszta przyjdzie w praktyce…

Konwersja typów prostych

JavaScript nie wymaga od ciebie abyś deklarował typ zmiennych. Przykładowo możesz utworzyć zmienną typu liczbowego o nazwie np. someVar, a następnie przypisać jej wartość znakową:

```
var someVar = 10;
someVar = "to jest napis";
```

Jeżeli byśmy chcieli dodać do zmiennej typu znakowego zmienną typu liczbowego wówczas otrzymalibyśmy wynik typu znakowego:

```
var someVar = "to jest napis " + 20;
console.log(someVar); //zwróci "to jest napis 20"

var someVar = "20" + 1;
console.log(someVar);  //zwróci "201"
```

Gdy od zmiennej typu znakowego w której skład wchodzą tylko znaki cyfr odejmiemy zmienną typu liczbowego wówczas wykonamy normalne równanie:

```
var someVar = "21" - 1;
console.log(someVar); //zwróci 20
```

Gdy od zmiennej typu znakowego w której skład wchodzą nie tylko znaki cyfr ale i litery odejmiemy zmienną typu liczbowego wówczas otrzymanym wynikiem będzie NaN (Not-A-Number)

```
var someVar = "20a" - 1;
console.log(someVar); //zwróci NaN
```

Podobna zasada działa w drugą stronę. Jeśli od zmiennej typu znakowego odejmiemy zmienną typu liczbowego, to w zależności czy pierwsza someVar zawiera litery czy też nie - otrzymamy wynik liczbowy lub typu NaN.

W praktyce jest to proste i łatwe…

Operatory

Operator	Nazwa działania	Równanie	Wynik	
+	Dodawanie	x = y + 2	y = 5	x = 7
-	Odejmowanie	x = y - 2	y = 5	x = 3
*	Mnożenie	x = y * 2	y = 5	x = 10
/	Dzielenie	x = y / 2	y = 5	x = 2.5
%	Reszta z dzielenia	x = y % 2	y = 5	x = 1
++	Inkrementacja	x = ++y	y = 6	x = 6
		x = y++	y = 6	x = 5
--	Dekrementacja	x = --y	y = 4	x = 4
		x = y--	y = 4	x = 5

Operator Przypisania

```
var myVar = 'Przykładowy tekst';
```

Operatory Porównania

Operator	Opis	Równanie	Zwróci
==	równe	x == 8	false
!=	różne	x != 8	true
>	większe od	x > 8	false
<	mniejsze od	x < 8	true
>=	większe bądź równe od	x >= 8	false
<=	mniejsze bądź równe od	x <= 8	true

Operatory Logiczne

Operator	Opis	Przykład	Wynik
&&	and (i)	(x < 10 && y > 1)	Prawda, bo x jest mniejsze od 10 i y jest większe od 1
||	or (lub)	(x > 8 || y > 1)	Prawda, bo x nie jest większe od 8, ale y jest większe od 1
!	not (negacja)	!(x == y)	Prawda, bo negujemy to, że x == y

Instrukcje warunkowe

If

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

```
if (warunek) {
   ...instrukcje jeżeli warunek jest poprany
}
```

lub

```
if (warunek) {
   ...instrukcje jeżeli warunek jest poprany
}
else{
   ...instrukcje jeżeli warunek nie jest poprany
}
```

If

Jeszcze jedna forma:

```
if (warunek1) {
   ...instrukcje jeżeli warunek1 jest poprany
}
else if(warunek2) {
   ...instrukcje jeżeli warunek1 nie jest poprany
}

else{
   ...instrukcje jeżeli warunek1 i warunek2 nie są poprany
}
```

Instrukcja switch

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

```
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 - nie potrzebuje break;
}
```

Pętle

Struktura pętli **while** ma następującą postać:

```
while (wyrażenie) {
   ...fragment kodu który będzie powtarzany...
}
```

Fragment kodu będzie powtarzany dopóki będzie spełniony warunek testowany w nawiasach. Aby pętla miała swój koniec, musimy w odpowiednim momencie sprawić, że testowany warunek zwróci wartość false.

Pętla typu do ... while

Podobnym rodzajem pętli jest pętla typu do ... while. Zasadniczą różnicą między tym typem a poprzednim jest to, że kod który ma być powtarzany zostanie wykonany przed sprawdzeniem wyrażenia.

Ogólna postać tego typu pętli jest następująca:

```
do {
   ...fragment kodu który będzie powtarzany...
} while (wyrażenie)
```

Pętla typu for

Kolejnym rodzajem pętli jest pętla typu for. Jest to najczęściej używany rodzaj pętli.

```
for (zainicjowanie zmiennych; wyrażenie testujące; zmiana wielkości zmiennej) {
   kod ktory zostanie wykonany pewną ilość razy
}
```

Przykład:

```
for (var x=0; x<100; x++) {
   console.log('Nie będę rozmawiał na lekcji Informatyki.');
}
```

Funkcje

Funkcja jest wywoływana przez inną część skryptu (w realnym życiu przez żonę), a w momencie jej wywołania zostaje wykonywany kod w niej zawarty.

Ogólna deklaracja funkcji ma następującą postać:

```
function nazwaFunkcji() {
   ...kod funkcji
}
nazwaFunkcji(); //wywołujemy funkcję
                                                    //lub
var nazwaFunkcji = function() {
   ...kod funkcji anonimowej
}
nazwaFunkcji(); //wywołujemy funkcję
```

Różnice w deklaracji funkcji

Podałem dwie metody deklaracji funkcji. Pierwsza z nich to tak zwana deklaracja funkcji. Deklaracja taka wymaga podania nazwy funkcji:

```
//deklaracja funkcji
function myFunction() {
   ...kod funkcji
}
```

Drugi sposób zwie się wyrażeniem. Anonimową funkcję (czyli taką, która nie ma nazwy) od razu podstawiamy pod zmienną:

```
//function jako wyrażenie (expresion)
var myFunction = function() {
   ...kod funkcji anonimowej
}
```

Różnice w deklaracji funkcji

Różnią się one nie tylko sposobem zapisu, ale także tym, jak taki kod jest interpretowany 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.

```
//Tutaj jest ok
myFunction();

function myFunction() {
   console.log('...');
}
```

```
//Błąd
myFunction();

var myFunction = function() {
   console.log('...');
}
```

Parametry (argumenty) funkcji

Częstokroć konieczne będzie przekazanie do funkcji określonych danych, które następnie zostaną przez nią przetworzone.

Parametry przekazuje się wypisując je między nawiasami występującymi po nazwie funkcji:

```
function calculate(number1, number2) {
    console.log(number1 + ' + ' + number2 + ' = ' + (number1 + number2))
    }

calculate(3, 4); //wypisze "3 + 4 = 7"
calculate(7, 7); //wypisze "7 + 7 = 14"
calculate(100, 100); //wypisze "100 + 100 = 200"
```

Instrukcja return

Każda funkcja po zakończeniu działania zwraca jakąś wartość. Może to być jakiś tekst, liczba czy wartość logiczna (prawda / fałsz).

Dzięki zastosowaniu instrukcji return możemy nakazać funkcji zwracanie określonej wartości. Instrukcja ta równocześnie przerywa dalsze działanie funkcji:

```
function calculate(number1, number2) {
    var result = number1 + number2;
    return result;
    }

console.log( calculate(10,4) ); //wypisze 14
```

Return – przykład:

```
function isEven(number) {
        if (number % 2 == 0) {
                return true;
        } else {
                return false;
        }
}


var x = 1;
if ( isEven(x) ) {
         console.log('Liczba' + x + ' nie jest parzysta');
        } else {
        console.log('Liczba' + x + ' jest parzysta');
        }
```

Zasięg zmiennych

W przypadku funkcji dochodzi dodatkowe pojęcie - zasięgu zmiennych. Do tej pory posługiwaliśmy się zmiennymi globalnymi, które były dostępne dla całego skryptu, wszystkich funkcji i w ogóle całego świata. Rozważmy prosty przykład:

Funkcja sum() nie tylko dokonuje wyliczenia sumy liczby, ale także modyfikuje naszą zmienną globalną x.

Pisząc kod funkcji powinniśmy się starać, by była ona zamkniętym bytem. Podajemy do niej jakieś parametry, funkcja to przetwarza za pomocą swojego kodu i zmiennych, po czym kończy swoje działanie zwracając jakiś wynik. Stosując zmienne globalne nie jesteśmy w stanie tego uzyskać, bo każdorazowo zmieniamy zmienne spoza funkcji.

```
var x = 10;

function sum(y) {
    x = x + y;
    return x;
}

console.log( sum(10) ); //wypisze 20
console.log( x ); //wypisze 20
```

Zasięg zmiennych

W Javascript możemy także korzystać także ze zmiennych lokalnych, które są dostępne tylko we wnętrzu danej funkcji. Aby zadeklarować taką zmienną wewnątrz funkcji używamy słowa var:

Dzięki stosowaniu zmiennych lokalnych nie tylko unikamy zmian w zmiennych globalnych, ale także unikamy problemu z nazewnictwem zmiennych (co pokazuje powyższy przykład).

```
var x = 1;

function showX() {
    var x = 2; //wewnętrzna zmienna x dostępna tylko w tej funkcji
    console.log(x); //wypisze 2
}

showX();
console.log(x); //wypisze 1 - jesteśmy poza funkcją
```

Tablice

Tworzenie nowej tablicy

Nic trudnego. Wystarczy skorzystać z poniższej instrukcji:

```
//deklaracja za pomocą literału
var ourTable = ['Marcin', 'Ania', 'Agnieszka'];
```

lub

```
var ourTable = new Array(6); //tworzy pustą tablicę o długości 6
```

Właściwość length

Każda tablica udostępnia nam właściwość length dzięki której możemy określić długość tej tablicy (ilość elementów). Dzięki temu możemy poznać index ostatniego elementu oraz w łatwy sposób przeprowadzać pętlę po wszystkich elementach naszej tablicy.

```
var ourTable = ['Marcin', 'Ania', 'Agnieszka', 'Piotrek', 'Grześ', 'Magda'];

console.log(ourTable.length); //6
```

Aby odwołać się do ostatniego elementu musimy odjąć 1 od liczby elementów (bo indexowanie jest od 0):

```
console.log( ourTable[ourTable.length-1] );
```

Pętla po tablicy

Aby zrobić pętlę po tablicy skorzystamy z jednej z dwóch pętli:

```
var ourTable = ['Marcin', 'Ania', 'Agnieszka', 'Piotrek', 'Grześ', 'Magda'];

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

String

String - teksty

W javascript tak samo jak w innych językach możemy operować nie tylko na liczbach, ale także na tekstach. Tekst powinien być zawarty w cudzysłowy lub apostrofy:

```
var text = "Ala ma kota, a kot ma Ale.";

console.log( text.length ); //pobieram długość tekstu - wypisze 26


text = text + "Ala go lubi, a kot ją wcale...";

console.log(text); //wypisze "Ala ma kota, a kot ma Ale. Ala go lubi, a kot ją wcale..."
```

KONIEC