Tworzenie zmiennych

```
var nazwaZmiennej;
```

Poprawna nazwa zmiennej może zaczynać się od znaków: `$ _ wielkich i małych liter`, a następnie można skorzystać ze znaków: `$ _ wielkich i małych liter` oraz `cyfr`.

Wyrazy zarezerwowane w języku, których nie powinno się używać jako nazw zmiennych				
abstract	default	float	long	switch
arguments	delete	for	native	synchronized
boolean	do	function	new	this
break	double	goto	null	throw
byte	else	if	package	throws
case	enum	implements	private	transient
catch	eval	import	protected	true
char	export	in	public	try
class	extends	instanceof	return	typeof
const	false	int	short	var
continue	final	interface	static	void
debugger	finally	let	super	volatile
while	with	yield		

Oprócz słów samego języka, nie powinno się korzystać także ze słów kierujących na poszczególne obiekty w Obiektowym Modelu Dokumentu.
Więcej znajdziesz tutaj: bit.ly/kurs-js-reserved-words

Tekstowy typ danych

Wartość tekstowa musi być zawarta w parze cudzysłowów lub apostrofów, np. `“Jan Kowalski”` lub `‘Jan Kowalski’`.

Wartość ta jest wartością prymitywną, lecz przy korzystaniu na niej z właściwości lub metod, jest w locie konwertowana do obiektu `String`, który zawiera następujące właściwości i metody:

Właściwość / metoda	Typ zwracanej wartości	Opis
`length`	number	zwraca ilość znaków w ciągu
`charAt(index)`	string	zwraca znak pod podanym indeksem `index`
`indexOf(str)`	number	zwraca indeks, pod którym znaleziono podany podciąg `str` lub `-1`, jeśli tekst go nie zawiera
`replace(str, newStr)`	string	zwraca nowy string, w którym `str` zostało zamienione na `newStr`
`slice(begin, end)`	string	zwraca nowy string, wycięty z oryginalnego, gdzie `begin` to indeks znaku, od którego należy zacząć wycinanie, a `end` to indeks, do którego wycinać (zwrócony string nie zawiera znaku spod indeksu `end`)
`split(str)`	array	zwraca tablicę z podciągami, które zostały utworzone przez rozdzielenie stringu na podanym w `str` znaku lub ciągu znaków; jeśli pod `str` przekazany zostanie pusty ciąg znaków `“”` to tablica zawierać będzie wszystkie znaki oryginalnego ciągu, każdy w osobnej komórce
`substr(begin, n)`	string	zwraca nowy string, podobnie jak metoda `slice` z tą różnicą, że parametr `n` oznacza ilość znaków do wycięcia, a nie indeks “do”
`toLowerCase()`	string	zwraca nowy string, z wszystkimi literami zamienionymi na małe
`toUpperCase()`	string	zwraca nowy string, z wszystkimi literami zamienionymi na wielkie

To jednak nie wszystkie metody dostępne na obiekcie `String`. Dokładny opis powyższych metod i wszystkich parametrów, jakie mogą przyjmować, a także opis pozostałych metod obiektu String znajdziesz tutaj: http://bit.ly/kurs-js-string

Liczbowy typ danych

Wartość liczbową wpisywana jest bezpośrednio, bez cudzysłowów czy apostrofów, np. `6` lub jako wartość zmiennoprzecinkową `6.23`

Istnieje także kilka specjalnych wartości numerycznych jak `Infinity` oraz `-Infinity`, reprezentujące "plus nieskończoność" i "minus nieskończoność", a także specjalna wartość `NaN`, która informuje, że wynik obliczeń to "Not a Number" (nie liczba).

Wartość liczbowa jest wartością prymitywną, lecz przy korzystaniu na niej z właściwości lub metod, jest w locie konwertowana do obiektu `Number`, który zawiera następujące właściwości i metody:

Właściwość / metoda	Typ zwracanej wartości	Opis
`toFixed(digits)`	string	zwraca liczbę z określoną parametrem `digits` liczbą cyfr po przecinku jako wartość tekstową
`toPrecision(digits)`	string	zwraca liczbę z określoną parametrem `digits` liczbą wszystkich cyfr jako wartość tekstową
`toString()`	string	zwraca liczbę zapisaną jako string

To jednak nie wszystkie metody dostępne na obiekcie `Number`. Dokładny opis powyższych, a także pozostałych metod obiektu `Number` znajdziesz tutaj: http://bit.ly/kurs-js-number

Wartości prawdziwe i fałszywe

Wartość prawda lub fałsz wpisywana jest bezpośrednio jako słowo `true` lub `false`.
Wartości te wykorzystywane są głównie w instrukcjach warunkowych, lecz w języku JavaScript oprócz wartości prawda/fałsz także inne wartości uznawane są za prawdziwe lub fałszywe.

Poniższe wartości uznawane są za fałszywe (wszystkie inne będą w instrukcjach warunkowych uznawane za prawdziwe):

`false`
`0`
`“”`
`null`
`undefined`
`NaN`

Wartości `true` i `false` są wartościami prymitywnymi, lecz również mają swoją reprezentację obiektową, którą jest konstruktor `Boolean`.
Więcej na ten temat znajdziesz tutaj: http://bit.ly/kurs-js-boolean

Operatory arytmetyczne

Operatory arytmetyczne służą do wykonywania działań na liczbach. Dostępne operatory to:

`+` dodawanie
`-` odejmowanie
`*` mnożenie
`/` dzielenie
`%` modulo (reszta z dzielenia)

Operator `+` ma jednak specjalne znaczenie dla ciągów znaków, które konkatenuje (łączy).

Kolejność działań matematycznych można zmieniać, grupując wyrażenia w nawiasy, np. `(2 + 2) * 4`

Powyższe operatory mogą być również zastosowane ze znakiem = w celu przypisania, np. zmienna += 2 jako alternatywa dla zmienna = zmienna + 2

Ich zastosowanie wygląda w tym przypadku następująco:

`+=`
`-=`
`*=`
`/=`
`%=`

Referencję operatorów arytmetycznych znajdziesz tutaj: http://bit.ly/kurs-js-arithmetic-ops

Operatory porównania

Operatory porównania pozwalają operować na każdej dostępnej w JavaScript wartości. Ich użycie zawsze zwraca wartość `true` lub `false`.

Operator	Przykład użycia	Opis
==	`2 == "2"`	porównuje wartość, nie typy; w tym przypadku zwróci `true`
===	`2 === "2"`	porównuje zarówno wartość jak i typ; w tym przypadku zwróci `false`
!=	`"Piotr" != "Tomasz"`	porównuje wartość, nie typy; sprawdza czy wartości nie są sobie równe; w tym przypadku zwróci `true`
!==	`"20" !== 20`	porównuje zarówno wartość jak i typ; sprawdza czy wartości nie są sobie równe; w tym przypadku zwróci `true`
>	`10 > 5`	porównuje wielkość liczb; w tym przypadku zwróci `true`; użyty na stringach porównuje je alfabetycznie, gdzie `"b" > "a"` zwróci `true`
<	`5 < 2`	porównuje wielkość liczb; w tym przypadku zwróci `false`; użyty na stringach porównuje je alfabetycznie, gdzie `"a" < "b"` zwróci `true`
>=	`10 >= 10`	porównuje wielkość liczb na zasadzie "większa niż lub równa"; w tym przypadku zwróci `true`; działa również na stringach
<=	`2 <= 5`	porównuje wielkość liczb na zasadzie "mniejsza niż lub równa"; w tym przypadku zwróci `true`; działa również na stringach

Bardzo ważne jest iż wszystkie wartości oprócz prymitywnych (`"Jacek"`, `20`, `true`, `undefined`, `null`) są porównywane operatorami `==`, `===`, `!=` oraz `!==` nie przez wartość, a przez referencję, a więc nie możemy sprawdzić np. czy dana tablica zawiera takie same elementy jak inna, porównując je w ten sposób `arr1 === arr2`

`true` zostanie zwrócone wyłącznie wtedy, gdy obie powyższe zmienne będą kierować na dokładnie tę samą tablicę.

Referencję operatorów porównania znajdziesz tutaj: http://bit.ly/kurs-js-comparsion-ops

Operatory logiczne

Operatory logiczne pozwalają określić czy jedna lub wszystkie z podanych wartości są wartościami prawdziwymi. Oprócz tego istnieje także operator zaprzeczenia (negacji), który zamienia wartość prawdziwą na fałsz lub wartość fałszywą na prawdę.

Operator	Przykład użycia	Opis
&&	`[1, 2] && “”`	logiczne AND; jeśli wartość po lewej stronie jest fałszywa, jest od razu zwracana; jeśli nie, sprawdzana jest wartość po prawej stronie i zostaje ona zwrócona zarówno jeśli jest prawdziwa jak i fałszywa; operator ten pozwala sprawdzić czy wszystkie podane wartości są prawdziwe jednak nie zwraca `true` lub `false`, a po prostu jedną z tych wartości; w przykładzie obok zwróci pusty ciąg znaków, który w instrukcji warunkowej `if` będzie uznawany za fałsz
||	`0 || 20`	logiczne OR; jeśli wartość po lewej stronie jest prawdziwa, jest od razu zwracana; jeśli nie, sprawdzana jest wartość po prawej stronie i zostaje ona zwrócona zarówno jeśli jest prawdziwa jak i fałszywa; operator ten pozwala sprawdzić czy przynajmniej jedna z podanych wartości jest prawdziwa jednak nie zwraca `true` lub `false`, a po prostu jedną z tych wartości; w przykładzie obok zwróci `20`, która w instrukcji warunkowej `if` będzie uznawana za prawdę
!	`!””`	logiczne NOT; operator ten odwraca wartość, która znajduje się po jego prawej stronie; `true` zamieni na `false` i odwrotnie; w przykładzie obok, wartość fałszywą jaką jest pusty ciąg znaków zamieni na `true`

Referencję operatorów logicznych znajdziesz tutaj: http://bit.ly/kurs-js-logical-ops

Inkrementacja i dekrementacja

Inkrementacja i dekrementacja pozwalają zmienić wartość zmiennej o 1. Inkrementacja zwiększa wartość, a dekrementacja ją zmniejsza. Wspomniane akcje rozróżniane są na 2 typy: postinkrementację i preinkrementację oraz postdekrementację i predekrementację.

Akcja	Przykład użycia	Opis
postinkrementacja	`zmienna++`	wartość zmiennej zostanie zwrócona, a dopiero później zostanie zwiększona o 1; jeśli zmiana wartości zmiennej nie jest pożądana, należy napisać `zmienna + 1`, a wtedy jej wartość zostanie zsumowana z jedynką bez zmiany tej wartości
preinkrementacja	`++zmienna`	wartość zmiennej zostanie zwiększona o 1, a następnie zwrócona;
postdekrementacja	`zmienna--`	wartość zmiennej zostanie zwrócona, a dopiero później zostanie zmniejszona o 1; jeśli zmiana wartości zmiennej nie jest pożądana, należy napisać `zmienna - 1`, a wtedy podstawiona zostanie różnica jej wartości i jeden
predekrementacja	`--zmienna`	wartość zmiennej zostanie zmniejszona o 1, a następnie zwrócona

Referencję operatorów inkrementacji i dekrementacji znajdziesz tutaj: http://bit.ly/kurs-js-increment

Instrukcje warunkowe

Instrukcje warunkowe służą do podejmowania decyzji o tym, który blok kodu się wykona, zależnie od wybranych czynników.

Instrukcja if

```
if(wartosc_prawdziwa) {
      // blok kodu do wykonania
}
```

Jeśli przekazana wartość nie będzie prawdziwa, możliwe jest użycie bloku `else` (w przeciwnym wypadku):

```
if(wartosc_prawdziwa) {
      // blok kodu do wykonania
} else {
      // blok kodu do wykonania
}
```

Po słowie `else` może pojawić się kolejna instrukcja warunkowa `if`, która sprawdzi kolejny warunek:

```
if(wartosc_prawdziwa) {
      // blok kodu do wykonania
} else if(kolejna_wartosc_prawdziwa) {
      // blok kodu do wykonania
} else {
      // blok kodu do wykonania
}
```

Skrócony zapis if

Instrukcja `if` może być zapisana jako wyrażenie:

```
wartosc ? true : false
```

np. jeśli zmienna `wartosc` przechowuje niepusty ciąg znaków, zostanie on zapisany do zmiennej, w przeciwnym wypadku podstawiony zostanie string `“Nieznajomy”`

```
var imie = wartosc ? wartosc : “Nieznajomy”;
```

Instrukcja switch

```
switch(wartosc) {
      case 10:
            // kod do wykonania jesli wartosc === 10
            break;
      case 20:
            // kod do wykonania jesli wartosc === 20
            break;
      default:
            // kod do wykonania żaden case się nie sprawdził
}
```

Po słowie kluczowym `case` umieszczona powinna być wartość, jaką sprawdzamy wobec zmiennej `wartosc`. Jeśli wartość będzie się zgadzać, wykonany zostanie blok kodu od dwukropka do słowa kluczowego `break` (dowolna ilość linijek).

Referencje:
if http://bit.ly/kurs-js-if
skrócony zapis if http://bit.ly/kurs-js-ternary
switch http://bit.ly/kurs-js-switch

Pętle

Pętle służą do powtarzania danego bloku kodu tak długo, dopóki zadany warunek jest spełniony (zwraca prawdę lub wartość prawdziwą).

Pętla while

```
while(wartosc_prawdziwa) {
      // blok kodu do wykonania
}
```

W miejscu wartosc_prawdziwa może się znaleźć dowolne wyrażenie, które będzie sprawdzane za każdą iteracją pętli, np. `zmienna < 10;` Aby pętla nie była nieskończona, warunek musi w którymś momencie przestać zwracać prawdę lub wartość prawdziwą (można wewnątrz pętli zmieniać wartość zmiennej).

Pętla do while

```
do {
      // blok kodu do wykonania
} while(wartosc_prawdziwa);
```

Działa identycznie jak pętla `while` z tą różnicą, że podany blok kodu wykona się przynajmniej raz, nawet jeśli warunek nie będzie spełniony. W przypadku pętli `while`, jeśli warunek nie jest spełniony, blok kodu w ogóle się nie wykona.

Pętla for

```
for(inicjalizacja; warunek; zmiana_wartosci) {
      // blok kodu do wykonania
}
```

Pętla wykonuje kod tak długo, jak warunek zwraca prawdę lub wartość prawdziwą. W miejscu `inicjalizacja` może pojawić się deklaracja zmiennej i jej wartości, w miejscu `warunek` kod, który będzie wykonywany za każdą iteracją, a w miejscu `zmiana_wartosci` kod, który będzie wykonywany po każdej iteracji, np.

```
for(var i = 1; i <= 10; i++) {
      // blok kodu do wykonania 10 razy
}
```

Pętla for in

Pętla ta pozwala iterować po właściwościach i metodach obiektu.

```
var osoba = {
      imie: “Jan”,
      nazwisko: “Kowalski”,
      wiek: 40
};

for(var klucz in osoba) {
      // blok kodu do wykonania,
      // w którym zmienna klucz będzie miała
      // kolejno wartości: “imie”, “nazwisko” i “wiek”
}
```

Aby wyświetlić daną wartość w bloku kodu pętli, należy skorzystać z zapisu `osoba[klucz]`

Przerywanie lub kontynuacja pętli

Każda z powyższych pętli może być w dowolnym momencie przerwana lub kontynuowana, bez wykonywania dalszego kodu. Służą do tego polecenia `break` oraz `continue`

```
for(var i = 1; i <= 10; i++) {
      if(i % 2 === 0) continue;

      // blok kodu do wykonania, jeśli liczba nieparzysta
}
```

W powyższym przykładzie, polecenie `continue` przejdzie do kolejnej iteracji pętli bez wykonywania kodu umieszczonego dalej, jeśli wartość w zmiennej `i` będzie liczbą parzystą.

```
var arr = [0, 20, 33, 42, 58, 16];

for(var i = 0; i <= arr.length; i++) {
      if(arr[i] === 42) break;

      // blok kodu do wykonania
}
```

W powyższym przykładzie, polecenie `break` przerwie całkowicie iterowanie pętli w momencie, gdy zostanie spełniony warunek z instrukcji `if`, a więc element tablicy będzie miał wartość `42`. Wartości `58` oraz `16` nigdy nie zostaną pobrane, gdyż pętle nie będzie już dalej wykonywana.

Referencje:
while http://bit.ly/kurs-js-while
do while http://bit.ly/kurs-js-do-while
for http://bit.ly/kurs-js-for
for in http://bit.ly/kurs-js-for-in
continue http://bit.ly/kurs-js-continue
break http://bit.ly/kurs-js-break

Obiekty

Choć w JavaScript prawie wszystko jest obiektem, to istnieje również specjalna konstrukcja, która w innych językach programowania jest nazywana np. tablicą asocjacyjną. W języku JavaScript określana jest jako "object litaral". Każdy obiekt może zawierać właściwości i metody. Właściwości to dane przechowywane wewnątrz obiektu, a metody to zwykłe funkcje, które są w nim zapisane.

```
var o = {
    imie: "Jan",
    nazwisko: "Kowalski",
    wiek: 32,
    przywitajSie: function() {
        return this.imie + " " + this.nazwisko;
    }
}
```

W powyższym przykładzie właściwościami są `imie`, `nazwisko` i `wiek`, a metodą jest `przywitajSie`. W obiekcie przechowywane mogą być wszystkie typy danych dostępne w języku JavaScript, tak samo jak w zmiennych.

Właściwości i metody mogą być przypisywane i odczytywane z obiektu jeszcze na dwa sposoby:

```
o.imie = "Tomasz";
o["nazwisko"] = “Nowak";
```

Właściwości i metody mogą być usuwane za pomocą operatora delete w ten sposób:

```
delete o.imie;
```

Jeśli właściwość zostanie usunięta, operator ten zwraca `true`, w przeciwnym wypadku `false`.

Wszystkie obiekty w języku JavaScript (nie tylko tzw. "object literal") przekazywane są przez referencję:

```
var o = {
    imie: "Jan"
};

var o2 = o;
```

W powyższym przykładzie zarówno zmienna `o` jak i `o2` kierują na dokładnie ten sam obiekt. Nie został on w żaden sposób skopiowany do zmiennej `o2`.

Referencję obiektów znajdziesz tutaj: http://bit.ly/kurs-js-objects

Tablice

Tablice są to specjalne obiekty, które pozwalają przechowywać dowolny typ danych w sposób uporządkowany, indeksowany od zera.

Tablica może być utworzona na dwa sposoby:

```
var arr = [1, 2];
var arr2 = new Array(1, 2);
```

Powyższy zapis stworzy identyczne tablice, jednak pierwszy z nich jest preferowany. W przypadku tablic z jednym elementem, zapis `new Array(20)`, do którego przekażemy wyłącznie jeden argument, zwróci tablicę z pustymi komórkami w podanej ilości, a nie nową tablicę z jednym elementem (w tym przypadku `20`), a więc poniższe zapisy nie są tożsame:

```
var arr = [20];
var arr2 = new Array(20);
```

Każda tablica posiada właściwość `length`, która pozwala odczytać ilość jej elementów. Aby z kolei odczytać zawartość danej komórki, należy podać jej indeks, w ten sposób:

```
arr[4]
```

W powyższym przykładzie odczytana zostanie **piąta** komórka tablicy, gdyż są one indeksowane **od zera**.

W podobny sposób można przypisywać elementy do tablicy, np.

```
arr[5] = “Tomek";
```

Właściwość / metoda	Typ zwracanej wartości	Opis
`push(elem1, elemN)`	number	dodaje na koniec tablicy przekazane elementy, a także zwraca nową długość tablicy (`length`)
`unshift(elem1, elemN)`	number	dodaje na początek tablicy przekazane elementy, a także zwraca nową długość tablicy (`length`)
`shift()`	mixed	usuwa pierwszy element tablicy, a następnie do zwraca
`pop()`	mixed	usuwa ostatni element tablicy, a następnie go zwraca
`concat(elem1, elemN)`	array	zwraca nową tablicę, w której znajdują się wszystkie elementy tablicy, na której została wywołana metoda, plus przekazane w argumentach elementy, którymi mogą być zawórno tablice jak i dowolne inne elementy
`indexOf(elem, from)`	number	zwraca indeks, pod którym znajduje się szukana wartość `elem` lub `-1`, jeśli tablica nie zawiera takiej wartości; drugi parametr pozwala określić, od którego indeksu zacząć wyszukiwanie
`join(sep)`	string	zwraca nowy string, który zawiera wszystkie elementy tablicy połączone stringiem przekazanym w `sep`
`forEach(callback)`	undefined	wykonuje funkcję przekazaną w `callback` dla każdego elementu tablicy; funkcja `callback` przyjmuje 3 parametry, kolejno: wartość, indeks, tablicę na której operuje
`map(callback)`	array	zwraca nową tablice; działa podobnie jak metoda `forEach`, iteruje po całej tablicy, jednak w nowej tablicy znajdują się wartości zwrócone z funkcji `callback`, możemy w ten sposób np. pomnożyć każdą liczbę jednej tablicy przez 2 i zapisać je w nowej tablicy
`filter(callback)`	array	zwraca nową tablicę; działa podobnie jak metoda `forEach`, iteruje po całej tablicy, jednak w nowej tablicy znajdują się wyłącznie wartości, przy których funkcja `callback` zwróciła `true`; możemy w ten sposób sprawdzić jakąś wartość i zapisać ją w nowej tablicy zwracając `true` lub nie, zwracając `false`
`reverse()`	array	odwraca kolejność elementów w tablicy i zwraca do niej referencję; nie zwraca nowej tablicy, ale operuje na oryginalnej

Właściwość / metoda	Typ zwracanej wartości	Opis
`sort(callback)`	array	sortuje tablicę alfabetycznie; opcjonalnie można skorzystać z funkcji `callback`, która przyjmuje dwa kolejne elementy tablicy jako parametry i jeśli zwróci `-1` to element pierwszy powinien zostać wstawiony przed drugim, jeśli `1` to na odwrót, a jeśli `0` to kolejność elementów nie jest zamieniana
`slice(begin, end)`	array	zwraca nową tablicę, która zawiera elementy wycięte z oryginalnej tablicy; `begin` oznacza indeks, od którego należy zacząć wycinanie, a `end` indeks, gdzie skończyć; element pod indeksem `end` nie jest jednak zawierany w wyciętej porcji tablicy
`splice(begin, n)`	array	zwraca wycięte elementy jako tablicę, a także modyfikuje tablicę, na której została wykonana; `begin` oznacza indeks, od którego należy zacząć usuwanie elementów, a `n` to ilość elementów do usunięcia z tablicy

To jednak nie wszystkie metody dostępne dla tablic. Dokładny opis powyższych, a także pozostałych, znajdziesz tutaj: http://bit.ly/kurs-js-arrays

Funkcje

Funkcja to jedna z najważniejszych konstrukcji każdego języka programowania. Pozwala grupować kod, który może być później w dowolnym momencie wykonany poprzez wywołanie funkcji.

Funkcję w języku JavaScript można utworzyć na następujące sposoby:

```
function f() {
    // kod do wykonania
}

var f = function() {
    // kod do wykonania
}

var f = function fn() {
    // kod do wykonania
}
```

Różnica pomiędzy tymi zapisami jest taka, iż w pierwszym przypadku z funkcji będzie można skorzystać w kodzie nawet przed jej deklaracją, np:

```
// wywołanie funkcji
f();

// definicja funkcji
function f() {
    // kod do wykonania
}
```

W drugim przypadku, po przypisaniu funkcji anonimowej do zmiennej, nie będzie możliwe jej użycie przez jej definicją.

Trzeci sposób pozwala na nadanie funkcji nazwy (w tym przypadku `fn`). Dzięki temu możliwe jest odwołanie się do tej nazwy wewnątrz kodu tej funkcji i wywołanie samej siebie (tzw. rekurencja).

Każda funkcja może definiować parametry, pod którymi mogą być przekazywane do funkcji odpowiednie dane, np.

```
// definicja funkcji
function powitaj(imie, nazwisko) {
    return “Cześć ” + imie + “ “ + nazwisko;
}

// wywołanie funkcji
powitaj(“Jan”, “Kowalski”);
```

W powyższym przykładzie, po wywołaniu funkcji i przekazaniu argumentów, pod zmiennymi lokalnymi `imie` oraz `nazwisko` wewnątrz funkcji, będą znajdować się kolejno wartości “Jan” oraz “Kowalski”, a sama funkcja poleceniem `return` zwróci ciąg “Cześć Jan Kowalski”. Polecenie `return` umożliwia zwracanie dowolnej wartości z funkcji, dzięki czemu można wynik jej wykonania przypisać do zmiennej:

```
var rezultat = powitaj(“Jan”, “Kowalski”);
```

Każda funkcja posiada specjalną zmienną lokalną `arguments`, która jest podobna do tablic i przechowuje pod poszczególnymi indeksami wszystkie przekazane argumenty, nawet jeżeli funkcja nie została przygotowana do przyjmowania jakichkolwiek, np.

```
// definicja funkcji
function f() {
    return arguments;
}

// wywołanie funkcji
f(“Jan”, “Kowalski”);
```

W powyższym przykładzie funkcja zwróci obiekt tablicopodobny `[“Jan”, “Kowalski”]`

Bardzo ważny jest również zakres zmiennych, który jest ograniczany w języku JavaScript wyłącznie funkcją. Żadna zmienna zdefiniowana wewnątrz funkcji nie będzie widoczna na zewnątrz, natomiast funkcja ma dostęp do zmiennych, które były zdefiniowane w tym samym zakresie co ona sama, np.

```
var imie = “Tomasz”;

function f() {
    // funkcja ma dostęp do zmiennej imie
    var nazwisko = “Kowalski”;
}

// tutaj dostępna jest zmienna imie, ale niedostępna jest zmienna nazwisko
```

Wbudowane funkcje języka JavaScript:

Funkcja	Typ zwracanej wartości	Opis
`eval(code)`	mixed	funkcja ta pozwala wykonać kod przekazany jako `string` w parametrze `code`; nie zaleca się jej stosowania, jeśli nie jest to absolutnie konieczne
`isFinite(value)`	boolean	zwraca `true` lub `false`, sygnalizując czy przekazana wartość jest liczbą skończoną
`isNaN(value)`	boolean	zwraca `true` lub `false`, sygnalizując czy przekazana wartość nie jest wartością liczbową
`parseInt(string)`	number / NaN	zwraca liczbę całkowitą znalezioną na początku stringu lub `NaN`, jeśli nie można było "wyłuskać" liczby
`parseFloat(string)`	number / NaN	zwraca liczbę zmiennoprzecinkową znalezioną na początku stringu lub `NaN`, jeśli nie można było "wyłuskać" liczby
`encodeURI(URI)`	string	koduje adres URL przekazany jako `string`, zamieniając zastrzeżone znaki na ich dozwoloną reprezentację; funkcja ta zakłada, że przekazany został cały adres, a nie tylko tzw. query string **?imie=Piotr** itd.
`decodeURI(URI)`	string	działa odwrotnie do funkcji `encodeURI`
`encodeURIComponent(s)`	string	koduje adres URL przekazany jako `string`, zamieniając zastrzeżone znaki na ich dozwoloną reprezentację; funkcja ta zakłada, że przekazany został komponent tzw. query string np. **nazwisko=Kowalska Nowak** itd.
`decodeURIComponent(s)`	string	działa odwrotnie do funkcji `encodeURIComponent`

JavaScript zawiera jeszcze więcej wbudowanych funkcji, jednak są one ukryte pod różnymi obiektami jako ich metody. Powyższe funkcje są globalne i można z nich korzystać bezpośrednio.

Dokładny opis powyższych funkcji i wiele więcej znajdziesz tutaj:
http://bit.ly/kurs-js-functions

Obiekt Math

Obiekt `Math` dostępny w języku JavaScript zawiera w sobie wiele przydatnych właściwości, a także metod z zakresu matematyki. Oto najważniejsze z nich:

Właściwość / metoda	Typ zwracanej wartości	Opis
`Math.E`	number	liczba Eulera, w przybliżeniu `2.7182818`
`Math.LN2`	number	logarytm naturalny z dwóch, w przybliżeniu `0.693`
`Math.LN10`	number	logarytm naturalny z dziesięciu, w przybliżeniu `2.303`
`Math.PI`	number	liczba PI, w przybliżeniu `3.14159`
`Math.SQRT2`	number	wartość pierwiastka z dwóch, w przybliżeniu `1.414`
`Math.abs(n)`	number	zwraca wartość absolutną przekazanej liczby
`Math.ceil(n)`	number	zwraca przekazaną liczbę zaokrągloną w górę
`Math.floor(n)`	number	zwraca przekazaną liczbę zaokrągloną w dół
`Math.round(n)`	number	zwraca przekazaną liczbę zaokrągloną w zależności od wartości po przecinku
`Math.log(n)`	number	zwraca logarytm naturalny przekazanej liczby
`Math.log10(n)`	number	zwraca logarytm dziesiętny przekazanej liczby
`Math.sqrt(n)`	number	zwraca wartość pierwiastka z przekazanej liczby
`Math.random()`	number	zwraca losową liczbę z zakresu `0.0` do `1.0`

Dokładny opis powyższych właściwości i funkcji, a także pozostałe znajdziesz tutaj: http://bit.ly/kurs-js-math

Obiekt Date

Obiekt `Date` pozwala odczytać systemową datę lub tworzyć nowe obiekty daty i manipulować ich wartościami.

Aby pobrać aktualną datę zapisaną w systemie operacyjnym, wystarczy utworzyć nowy obiekt daty w ten sposób:

```
var data = new Date();
```

Pod zmienną `data` znajdzie się obiekt, który zawiera mnóstwo przydatnych metod, które zostały opisane w tabeli poniżej.

Konstruktor `Date` pozwala również na przekazanie mu jednego lub wielu argumentów. Jego wywołanie z jednym argumentem spowoduje utworzenie nowego obiektu daty i ustawienie jego czasu na datę **“ilość sekund od 1 stycznia 1970 roku”**, gdzie ilość sekund, to przekazany argument:

```
var data = new Date(1000000);
```

Konstruktor ten przyjmuje również kolejne argumenty i jeśli zostaną podane przynajmniej 2, traktowane są one następująco:

```
var data = new Date(rok, miesiac, dzien, godzina, minuty, sekundy, milisekundy);
```

Przy takim wywołaniu dwa pierwsze parametry są niezbędne, a każdy kolejny jest opcjonalny.

Najważniejsze metody dostępne na obiekcie `Date`:

Metoda	Typ zwracanej wartości	Opis
`getFullYear()`	number	zwraca rok w formacie XXXX
`getMonth()`	number	zwraca miesiąc jako liczbę, indeksując od zera (styczeń === `0`)
`getDate()`	number	zwraca dzień
`getDay()`	number	zwraca dzień tygodnia jako liczbę, np. `5` to piątek
`getHours()`	number	zwraca godzinę
`getMinutes()`	number	zwraca liczbę minut
`getSeconds()`	number	zwraca liczbę sekund
`getMilliseconds()`	number	zwraca liczbę milisekund
`getTime()`	number	zwraca liczbę sekund, które upłynęły od 1 stycznia 1970 roku
`toString()`	string	zwraca datę w komputerowym formacie

Bardzo ważne jest, iż powyższe metody z przedrostkiem **get**, mają również swoje odpowiedniki z przedrostkiem **set**, które pozwalają ustawiać odpowiednie wartości na obiekcie daty, np. `setFullYear(2020)` zmieni w dacie jedynie rok na **2020**.

Dokładny opis powyższych metod, a także wszystkie pozostałe znajdziesz tutaj: http://bit.ly/kurs-js-date

Obsługa wyjątków

Język JavaScript udostępnia mechanizm obsługi wyjątków/błędów, które mogą się pojawiać podczas wykonywania skryptów.

Do wyłapywania błędów służy specjalna instrukcja `try…catch`, która wygląda następująco:

```
try {
    // kod do wykonania
} catch(e) {
    // jeśli wystąpił błąd, jego obiekt
    // znajduje się pod zmienną e
}
```

W bloku `try` wstawić należy dowolny kod do wykonania, może to być np. wywołanie funkcji. Jeśli w tym bloku wystąpi błąd, np. odwołamy się do funkcji, która nie istnieje, to wykonywanie skryptu automatycznie przeskoczy do bloku `catch`, gdzie pod zmienną `e` znajdzie się obiekt z informacjami na temat błędu (`e.message`).

Blok `try…catch` przyjmuje również opcjonalny blok `finally`, który wykona się zawsze, niezależnie od tego czy wystąpi błąd:

```
try {
    // kod do wykonania
} catch(e) {
    // jeśli wystąpił błąd, jego obiekt
    // znajduje się pod zmienną e
} finally {
    // kod do wykonania zawsze
}
```

Błędy mogą być zgłaszane zarówno przez interpreter języka JavaScript jak i przez programistę. Własny błąd możemy zgłosić poleceniem `throw` w ten sposób:

```
function powitaj(imie) {
    if(!imie) {
        throw new Error("Imię nie zostało podane");
    } else {
        return "Cześć " + imie;
    }
}
```

Jeśli w bloku `try` wykonana zostanie powyższa funkcja, a parametr `imie` nie zostanie przekazany, zostanie zgłoszony błąd za pomocą polecenia `throw`. W tym przypadku tworzony jest nowy obiekt błędu (`Error`), któremu możemy przekazać wiadomość. Będzie ona widoczna w bloku `catch` pod `e.message`. Tworzenie nowego obiektu błędu jest dobrą praktyką, natomiast polecenie `throw` może zwrócić dowolną wartość, która znajdzie się pod zmienną `e` w bloku `catch`.

Więcej informacji znajdziesz tutaj: http://bit.ly/kurs-js-try-catch oraz tutaj: http://bit.ly/kurs-js-throw

Obiektowy Model Dokumentu

Obiektowy Model Dokumentu to głównie reprezentacja kodu HTML w postaci obiektów, z którymi za pomocą języka JavaScript możemy wchodzić w interakcję.

Z poziomu pisanych skryptów mamy dostęp do kilku ciekawych, globalnych obiektów:

`window` jest to najwyżej położony w hierarchii obiekt, pod którym znajduje się wiele metod oraz właściwości. Co ważne, wszystkie zmienne utworzone bez słowa kluczowego `var` lub w globalnym zakresie, stają się tak naprawdę właściwościami obiektu `window`.

Wewnątrz obiektu `window` znajduje się obiekt `navigator`, do którego można się odwołać poprzez `window.navigator` lub po prostu `navigator`. Obiekt ten zawiera informacje na temat klienta (np. przeglądarki internetowej).

Kolejnym ważnym obiektem jest `screen`, który zawiera informacje na temat ekranu użytkownika, takie jak rozdzielczość czy głębia kolorów. Można się do niego odwołać podobnie jak w przypadku obiektu `navigator`.

Ostatnim z ważnych obiektów jest `location`, który zawiera właściwości, a także metody związane z paskiem adresu przeglądarki. Pozwala on odczytać np. protokół lub przeładować stronę. Można się do niego odwołać poprzez obiekt `window` lub bezpośrednio.

Referencje:
window http://bit.ly/kurs-js-window
navigator http://bit.ly/kurs-js-navigator
screen http://bit.ly/kurs-js-screen
location http://bit.ly/kurs-js-location

setTimeout i setInterval

Globalny obiekt `window` udostępnia funkcje, które pozwalają opóźnić wykonywanie kodu lub wykonywać go w stałych odstępach.

Funkcja `setTimeout` przyjmuje jako pierwszy parametr funkcję do wykonania, a jako drugi liczbę milisekund, po których wykonać dany kod, np.

```
setTimeout(function() {
    // kod do wykonania jednokrotnie
}, 2000);
```

Funkcja `setInterval` przyjmuje dokładnie takie same parametry jak funkcja `setTimeout`, jednak wykonuje przekazaną funkcję regularnie co określony czas:

```
setInterval(function() {
    // kod do wykonywania do 2 sekundy
}, 2000);
```

Obie funkcje zwracają referencję do timera, która będzie potrzebna by anulować wykonywanie opóźnionego kodu za pomocą funkcji `clearTimeout` i `clearInterval`, w ten sposób:

```
var timer = setTimeout(function() {
    // kod do wykonania jednokrotnie
}, 2000);

clearTimeout(timer);

var timer = setInterval(function() {
    // kod do wykonywania do 2 sekundy
}, 2000);

clearInterval(timer);
```

Jeśli użyjemy funkcji `clearTimeout` przez upływem (w tym przypadku) dwóch sekund, to kod nigdy się nie wykona. Podobnie z funkcją `clearInterval`, która pozwoli przerwać wykonywany interwał i całkowicie anulować dalsze jego wykonywanie.

Więcej na ten temat znajdziesz tutaj: http://bit.ly/kurs-js-timers

Metody przeszukiwania drzewa DOM

Aby z poziomu skryptu wyszukać na stronie internetowej odpowiedniego elementów lub wielu elementów, należy skorzystać z jednej z dostępnych do tego celu metod.

Metoda	Opis
`getElementById(str)`	zwraca referencję do obiektu, który posiada atrubut `id` równy `str`
`getElementsByName(str)`	zwraca referencję do obiektu tablicopodobnego, który zawiera wszystkie znalezione elementy, które posiadają atrybut `name` równy `str`
`getElementsByTagName(str)`	zwraca referencję do obiektu tablicopodobnego, który zawiera wszystkie znalezione elementy, które pasują do tagu `str` np. `“div”`
`getElementsByClassName(str)`	zwraca referencję do obiektu tablicopodobnego, który zawiera wszystkie znalezione elementy, które posiadają klasę `str`
`querySelector(str)`	zwraca referencję do pierwszego znalezionego obiektu, który pasuje do przekazanego selektora CSS, np. `“#container”`
`querySelectorAll(str)`	zwraca referencję do obiektu tablicopodobnego, który zawiera wszystkie znalezione elementy, które pasują do przekazanego selektora CSS
`document.all`	skrót do wszystkich elementów w dokumencie
`document.forms`	skrót to wszystkich formularzy w dokumencie
`document.images`	skrót do wszystkich obrazów w dokumencie
`document.links`	skrót do wszystkich linków w dokumencie

Powyższe metody wykonywane są na dokumencie lub na innych elementach. Aby wyszukać na stronie wszystkich elementów z klasą `“red”`, możemy wpisać:

```
document.querySelectorAll(“.red”);
```

natomiast aby znaleźć je wyłącznie w wybranym elemencie div (zapisanym już do zmiennej `div`), możemy napisać:

```
div.querySelectorAll(“.red”);
```

Więcej na temat przeszukiwania drzewa DOM znajdziesz tutaj:
http://bit.ly/kurs-js-query-dom

Tworzenie nowych węzłów

DOM API umożliwia tworzenie nowych węzłów, które można następnie wstawić na stronę. Mogą to być zarówno węzły tekstowe jak i elementy HTML.

Nowy element HTML utworzyć można za pomocą metody createElement dostępnej na obiekcie dokumentu, w ten sposób:

```
var nowyDiv = document.createElement("div");
```

Do zmiennej zostanie zwrócony nowy element, na którym można pracować dokładnie tak samo jak na elementach wyszukanych w drzewie DOM. Nie jest on jednak nigdzie wstawiony.

Jeśli planujemy utworzyć kontener na wiele elementów, które mają być następnie wstawione na stronę, możemy skorzystać z fragmentu dokumentu:

```
var fragment = document.createDocumentFragment();
```

Następnie można do niego dodawać dowolną liczbę elementów, a dopiero później fragment wstawić na stronę.

Podobnie można utworzyć węzeł tekstowy, za pomocą odpowiedniej metody:

```
var tekst = document.createTextNode("Treść");
```

Węzeł tekstowy posiada swoje własne właściwości i metody. Może być również wstawiany do elementów HTML jako ich treść.

Referencje:
createElement http://bit.ly/kurs-js-createelement
createDocumentFragment http://bit.ly/kurs-js-createdocumentfragment
createTextNode http://bit.ly/kurs-js-createtextnode

Przypisywanie treści

Każdy element HTML, znaleziony na stronie lub utworzony na nowo, może posiadać treść. Przypisywać i odczytywać możemy ją na kilka sposobów.

```
div.innerHTML; // zwróci cały kod HTML zawarty w elemencie
div.outerHTML; // podobnie jak wyżej lecz razem z kodem elementu
div.textContent; // zwróci jedynie zawartość tekstową elementu
```

Powyższe metody mogą być również używane do przypisywania nowej treści do elementów HTML. Jeśli w przypisywanym stringu nie ma kodu HTML, to należy skorzystać z `textContent`. W przeciwnym wypadku z `innerHTML`, np.

```
div.textContent = “Nowa treść diva”;
div.innerHTML = “<strong>Nowa treść diva</strong>”;
```

Więcej znajdziesz tutaj: http://bit.ly/kurs-js-elems-content

Wstawianie i usuwanie elementów ze strony

DOM API udostępnia również szereg metod służących do wstawiania i usuwania elementów z drzewa dokumentu. Oto najpopularniejsze z nich:

Metoda	Opis
`appendChild(elem)`	wstawia przekazany element do elementu, na którym została wykonana; element jest wstawiony jako ostatnie dziecko
`insertBefore(elem, before)`	wstawia przekazany element do elementu, na którym została wykonana, lecz przed elementem, który został przekazany jako drugi parametr
`replaceChild(elem, replace)`	wstawia przekazany element do elementu, na którym została wykonana i zamienia go z elementem przekazanym jako drugi parametr
`removeChild(elem)`	usuwa przekazany element z elementu, na którym została wykonana

Więcej informacji znajdziesz tutaj: http://bit.ly/kurs-js-elems-inserting

Relacje między elementami

Między węzłami wystąpują relacje typu rodzic, dziecko lub rodzeństwo. Aby z poziomu wybranego węzła odwołać się do jego rodziny, należy skorzystać z następujących jego właściwości:

Właściwość	Opis
`childNodes`	zwraca wszystkie dzieci danego elementu (nie tylko elementy HTML, ale także np. węzły tekstowe)
`children`	zwraca wszystkie dzieci danego elementu (wyłącznie elementy HTML)
`firstChild`	zwraca pierwsze dziecko danego elementu (nie tylko element HTML, ale także np. węzeł tekstowy)
`firstElementChild`	zwraca pierwsze dziecko danego elementu (wyłącznie element HTML)
`lastChild`	zwraca ostatnie dziecko danego elementu (nie tylko element HTML, ale także np. węzeł tekstowy)
`lastElementChild`	zwraca ostatnie dziecko danego elementu (wyłącznie element HTML)
`nextSibling`	zwraca następny węzeł (nie tylko element HTML, ale także np. węzeł tekstowy)
`nextElementSibling`	zwraca następny węzeł (wyłącznie element HTML)
`previousSibling`	zwraca poprzedni węzeł (nie tylko element HTML, ale także np. węzeł tekstowy)
`previousElementSibling`	zwraca poprzedni węzeł (wyłącznie element HTML)
`parentNode`	zwraca rodzica danego elementu

Więcej informacji znajdziesz tutaj: http://bit.ly/kurs-js-elems-relationships

Praca z atrybutami

Każdy element HTML może posiadać atrybuty, przypisane w kodzie HTML lub dynamicznie z poziomu skryptu. Oto metody do pracy z atrybutami:

Metoda	Typ zwracanej wartości	Opis
`getAttribute(name)`	string	zwraca wartość podanego atrybutu
`setAttribute(name, v)`	undefined	ustawia atrybut lub zmienia jego wartość; `name` to nazwa atrybutu, a `v` to jego wartość
`hasAttribute(name)`	boolean	zwraca `true` lub `false`, testując czy element posiada wskazany atrybut

Na obiektach można również korzystać z licznych skrótów do atrybutów, np. `img.src` zamiast `img.getAttribute("src")` (podobnie w przypisywaniem).

Aby pobrać wszystkie atrybuty elementu w formie obiektu tablicopodobnego, możemy skorzystać z jego właściwości `attributes`.

Referencje:
getAttribute http://bit.ly/kurs-js-getattribute
setAttribute http://bit.ly/kurs-js-setattribute
hasAttribute http://bit.ly/kurs-js-hasattribute
attributes http://bit.ly/kurs-js-attributes

Praca z klasami CSS

Każdy obiekt HTML udostępnia właściwości i metody, które pozwalają odczytywać, ustawiać oraz zmieniać klasy CSS do niego przypisane.

Najprostszym sposobem na odczytanie wszystkich klas elementu, jest skorzystanie z jego właściwości `className`. W ten sposób można również przypisywać klasy do elementów, jednak przypisanie nowej wartości nadpisze wszystkie wcześniejsze klasy.

Z tego powodu lepiej skorzystać z obiektu `classList` i jego metod. Obiekt ten jest dostępny na każdym obiekcie HTML, np. `div.classList.add("klasa")`

Metoda	Opis
`classList.add(c, cN)`	dodaje jedną lub więcej klas do elementu
`classList.remove(c, cN)`	usuwa jedną lub więcej klas z elementu
`classList.toggle(c)`	dodaje klasę, jeśli jej nie ma lub usuwa, jeśli istnieje
`classList.contains(c)`	zwraca `true` lub `false`, sprawdzając czy klasa istnieje

Referencje:
className http://bit.ly/kurs-js-classname
classList http://bit.ly/kurs-js-classlist

Praca ze stylami CSS

Każdy obiekt HTML zawiera właściwość `style`, która kieruje na specjalny obiekt pozwalający pracować ze stylami CSS elementu.

Aby do elementu przypisać np. kolor, a więc właściwość CSS o nazwie `color`, należy skorzystać z następującego zapisu:

```
div.style.color = “#ff0000”;
```

W przypadku właściwości takich jak `font-size`, należy skorzystać z notacji camelCase lub z nawiasów kwadratowych:

```
div.style.fontSize = “16px”;
div.style[“font-size”] = “16px”;
```

Aby przypisać więcej styli jednocześnie, skorzystać można z właściwości `cssText` w ten sposób:

```
div.style.cssText = “color: #ff0000; font-size: 16px;”;
```

Odczytywać wartości poszczególnych właściwości można wyłącznie wtedy, gdy są one przypisane do atrybutu powyższymi metodami lub w HTMLu. Aby jednak odczytać np. `font-size` elementu, do którego nie została taka właściwość przypisana, należy skorzystać z metody obiektu `window`, która pozwala podejrzeć wyliczone style w ten sposób:

```
window.getComputedStyle(elem).fontSize;
```

gdzie `elem` to referencja do obiektu znalezionego na stronie lub nowo utworzonego.

Więcej informacji znajdziesz tutaj: http://bit.ly/kurs-js-style

Geometria i położenie elementów

Każdy obiekt HTML znajdujący się w dokumencie posiada właściwości takie jak położenie w przestrzeni strony czy szerokość oraz wysokość.

Aby odczytać położenie elementu względem rodzica, który jest kontekstem pozycjonującym (ma właściwość CSS position: relative/absolute/fixed) należy skorzystać z jego właściwości `offsetTop` oraz `offsetLeft`. Kontekst pozycjonujący dla danego elementu można sprawdzić poprzez `offsetParent`.

Powyższe wartości można również odczytać za pomocą obiektu pobranego przez metodę `getBoundingClientRect()`, a następnie `top`, `left`, `right` i `bottom`.

Za pomocą w/w obiektu można również odczytać wymiary elementu (właściwości `width` oraz `height`). Alternatywnie, można skorzystać z właściwości `offsetWidth` i `offsetHeight`. Zawierają one odpowiednio szerokość oraz wysokość elementu, wraz z paddingiem i borderem. Z kolei `clientWidth` oraz `clientHeight` zwracają wartości bez wliczania obramowania.

Jeśli treść elementu się w nim nie mieści i wyświetlane są suwaki, nadal można odczytać całkowitą szerokość i wysokość elementu “wewnątrz”, a więc z treścią, za pomocą właściwości `scrollWidth` oraz `scrollHeight`.

Powyżej opisane właściwości i metody działają dla wszystkich obiektów reprezentujących elementy HTML. Aby jednak odczytać szerokość i wysokość samego okna przeglądarki, należy skorzystać z właściwości `innerWidth` oraz `innerHeight` obiektu `window`.

Oprócz wymiarów elementów, odczytać lub przypisać można także pozycje suwaków. Główne suwaki widoczne na stronie wyświetlane są dla elementu `body`. Wartość przewinięcia w pikselach od góry odczytać można za pomocą właściwości `scrollTop`, a od lewej strony `scrollLeft` obiektu `body`. Wartości te mogą być również ustawiane, co spowoduje przeskoczenie suwaków do odpowiednich pozycji.

Przewinąć widok można również za pomocą metody `scrollTo(x, y)` obiektu `window`, gdzie `x` to pozycja suwaka poziomego, a `y` to pozycja suwaka pionowego. Alternatywnie działa metodą `scrollBy(x, y)`, która przewija jednak o określoną wartość względem aktualnej pozycji suwaków.

Więcej informacji znajdziesz tutaj: http://bit.ly/kurs-js-geometry

Zdarzenia

Niezwykle ważną funkcjonalnością DOM API jest możliwość przypisywania zdarzeń. Mogą być one przypisywane zarówno do elementów HTML, np. kliknięcie na jakiś przycisk, ale także dla najróżniejszych obiektów, np. otrzymanie odpowiedzi z serwera po wysłaniu żądania AJAX.

Istnieją 3 sposoby przypisywania zdarzeń. Pierwszy z nich to dodanie do elementu HTML atrybutu, np. `onclick` dla zdarzenia `click`, w ten sposób:

```
<button onclick="alert('Zostałem kliknięty!')">Kliknij mnie</button>
```

Wartość atrybutu to tekst zawierający kod JavaScript do wykonania. Nie jest to jednak zalecany sposób przypisywania jakichkolwiek zdarzeń.

Drugi sposób to przypisanie odpowiedniej właściwości dla obiektu. Załóżmy, że referencję do przycisku zapisaliśmy w zmiennej `button`:

```
button.onclick = function() {
    alert("Zostałem kliknięty!");
};
```

Podobnie jak w przypadku atrybutu, nazwa zdarzenia poprzedzana jest przedrostkiem "on". Wadą takiego rozwiązania jest fakt, że możemy przypisać wyłącznie jedną funkcję obsługi danego zdarzenia. Każde kolejne przypisanie nadpisze wcześniejsze.

Trzeci, najlepszy sposób, to skorzystanie z metody `addEventListener`:

```
button.addEventListener("click", function() {
    alert("Zostałem kliknięty!");
}, false);
```

Metoda ta jako pierwszy parametr przyjmuje rodzaj zdarzenia (bez przedrostka "on"), jako drugi funkcję do wykonania, a jako trzeci `true` lub `false` czy funkcja ma zostać wykonana w fazie capturing.

Zdarzenie można w dowolnym momencie usunąć z elementu. W dwóch pierwszych przypadkach wystarczy napisać

```
button.onclick = null;
```

W przypadku trzecim, należy skorzystać z metody `removeEventListener`, która przyjmuje identyczne parametry jak metoda `addEventListener`. Aby funkcję można było jednak usunąć, musi być ona wcześniej podawana przez referencję, a nie jak w powyższym przykładzie jako funkcja anonimowa. W takim przypadku nie ma możliwości jej usunięcia.

Funkcja obsługi zdarzenia może przyjmować parametr, który będzie zawierał obiekt zdarzenia:

```
button.onclick = function(e) {
    // zmienna e przechowuje obiekt Event
};
```

W obiekcie zdarzenia znajduje się bardzo wiele informacji na jego temat, a także kilka przydatnych metod. Najważniejsze z nich to `preventDefault`, `stopPropagation` oraz `stopImmediatePropagation`.

Pierwsza z nich zapobiegnie domyślnej akcji przeglądarki. Jeśli kliknięty został link, to dzięki temu przeglądarka nie przejdzie do nowej strony. Jeśli zdarzenie dotyczyło wysłania formularza, zostanie ono wstrzymane.

Metoda `stopPropagation` zapobiegnie wywoływaniu się zdarzenia na przodkach elementu np. klikniętego (jeśli przycisk znajdował się w elemencie `div`, to ten element również został kliknięty).

Ostatnia metoda, `stopImmediatePropagation` zapobiegnie z kolei wykonywaniu się kolejnych funkcji obsługi zdarzeń dla tego samego elementu.

Więcej informacji znajdziesz tutaj: http://bit.ly/kurs-js-events
A listę dostępnych zdarzeń tutaj: http://bit.ly/kurs-js-events-list