Podstawowy kurs programowania w JavaScript

Marcin Kałużny
CODE:ME

Wydanie pierwsze

Zmienne
Krótki wstęp

Specyfikacja języka programowania JavaScript do standardu ECMAScript5 pozwalała zadeklarować zmienne poprzez operator `var`. Deklaracja przez ten operator może doprowadzić do nieporządanych efektów działania kodu.

Specyfikacja ECMAScript6 wprowadza nowy operator `let`, który ma zastąpić deklarowanie zmiennych przez operator var. Z tego względu bardzo często mówi się, że operator `let` to nowy var.

ECMAScript6 dostarcza nam jeszcze jednego operatora `const` do deklarowania tz. stałych, ale jak się okazuje nie do końca stałe będą stałymi - o tym poniżej.

Czym są zmienne

Podstawową i jedyną funkcją zmiennych jest przechowywanie danych. Dane te można zmieniać na co wskazuje nazwa „zmienne". Zmienną możemy zastosować również wtedy kiedy wartość danych nie jest znana podczas pisania kodu, a jest wynikiem działania przeprowadzonego po uruchomieniu programu.

Zmienne w JavaScript nie wymagają inicjalizacji (przypisania wartości). Są inicjalizowane niejawnie. Zmienna deklarowane przy użyciu `var` lub `let`, której nie przypiszemy początkowej wartości przyjmie automatycznie wartość `undefined`.

```
let mojaZmienna;
```

Zmienne mogą przechowywać różne typy wartości w różnych momentach. Wynika to z faktu, że **zmienne w JavaScript nie mają ścisłych typów** - uważam że to jest zaletą języka JavaScript.

Nazwy zmiennych mogą składać się z kombinacji liter, cyfr, oraz znaków podkreślenia lub $. Nie można jednak zacząć od cyfry.

```
// ŹLE
let 1mojaNazwa = 10;

// DOBRZE
let mojaZmienna1 = 10,
    _mojaZmienna1 = 11,
    moja_zmienna1 = 12,
    $mojaZmienna = 13;
```

Wielkość liter ma znaczenie

W nazwach zmiennych rozróżniane są wielkie i małe litery.

Zmienne nie mogą być nazwane jak słowa zarezerwowane.

Zasięg zmiennych

Zmienną, która została zadeklarowana poza funkcją nazywamy *globalną*. Jest ona dostępna z każdego miejsca w naszym kodzie.

Zmienną, która została zadeklarowana wewnątrz funkcji nazywamy *lokalną*, ponieważ można jej użyć tylko w ciele funkcji.

ECMAScript 6 wprowadza zasięg blokowy czyli zmienne zadeklarowane przez `let` w danym bloku { } są ograniczone do wnętrza tego bloku

```
if (true) {
    var x = 5;
}


if (true) {
    let y = 5;
}


console.log(x, y);
```

var

Deklarowanie zmiennych przez ten operator może doprowadzić do pojawienia się błędów w kodzie napisanym przez niedoświadczonego programisty. Dlaczego? Odpowiedz jest prosta deklaracja zmiennej przez operator `var` nie ogranicza utworzonej przez niego zmiennej do bloku, a jedynie do przestrzeni całej funkcji.

> ! *Zakres zmiennych zadeklarowanych przez operator `var` w bloku nie ogranicza się tylko do tego bloku. Zakresem jest cała funkcja.*

let

Zmienne zadeklarowane przez operator `let` mają zasięg blokowy.

Zmienne zadeklarowane przez operator `let` w przestrzeni globalnej (window - w BOM) nie są własnościami tego obiektu. Są aktywne w zakresie niewidzialnego bloku, który hipotetycznie obejmuje cały kod.

Błędem jest użycie zmiennej utworzonej przy pomocy `let` przed jej zadeklarowaniem.

Ponowna deklaracja zmiennej za pomocą operatora `let` powoduje błąd składni

const

Deklaruje zmienną stałą tylko do odczytu. Zmienna nie może zostać ponownie zadeklarowana lub przyjąć nowej wartości kiedy działa nasz skrypt. Możemy to zrobić jedynie przy jej inicjalizacji.

Zasady zasięgu są dla niej takie same jak dla zmiennych tworzonych z użyciem `let`.

Limitacją dla stałych jest fakt, że nie możemy zadeklarować takiej, która wykorzystuje nazwę występującą już jako identyfikator zmiennej lub funkcji w tym samym bloku.

Typy proste i referencje

Czym są typy?

W JavaScript funkcjonują dwa rodzaje typów danych:

- typy proste (ang. *primitives*), które służą do zapisywania danych w prostej postaci
- typy referencyjne (ang. *references*) gdzie dane są zapisywane w obiektach, gdzie do ich lokalizacji w pamięci prowadzi referencja.

W JavaScript zmienna jest zapisywana w tz. obiekcie zmiennych. Proste wartości są przechowywane bezpośrednio w tym obiekcie, a w przypadku referencji w obiekcie zmiennych jest zapisywany wskaźnik do lokalizacji obiektu w pamięci.

Standard ECMAScript definiuje typ jako zbiór wartości, a każdy z typów prostych definiuje zakres wartości, jakie typ może przybierać oraz literową reprezentację tego typu.

Typy proste

W JavaScript występuje pięć typów prostych:

boolean	- wartość logiczna `true` albo `false`
number	- liczbowa wartość całkowita lub zmiennoprzecinkowa
string	- znak lub łańcuch znaków
null	- ma tylko jedną wartość `null`
undefined	- wartość zmiennej, która jeszcze nie została zainicjalizowana

Wartości wszystkich typów prostych mają reprezentację w postaci literałów. Są to te wartości, które nie znajdują się w zmiennych.

Operator `typeof`

Operator typeof pobiera jeden parametr - zmienną lub wartość do sprawdzenia, np.

```
let s = 'testujemy';

alert(typeof s);
alert(typeof 100);
```

Boolean (true i false)

Do typu boolean należą dwie wartości `true` i `false`. Używa się ich bez cudzysłowów. Jeżeli `true` lub `false` umieścimy w cudzysłowie to będzie on łańcuchem znaków.

```
console.log(typeof true);
console.log(typeof 'false');
```

Liczby (number)

JavaScript ma tylko jeden typ liczbowy. Wewnętrznie jest on reprezentowany jako 64-bitowa liczba zmiennoprzecinkowa, tak samo jak double w Javie. Nie ma osobnego typu dla liczb całkowitych. Dzięki takiemu rozwiązaniu nie pojawiają się problemy przepełnienia krótkich typów całkowitych, a wiedza o liczbie ogranicza się w JavaScript, tylko do tego, że jest to liczba. Błędy numeryczne zostają automatycznie wyeliminowane.

Liczby binarne zmiennoprzecinkowe nie nadają się zbyt dobrze do operacji na ułamkach dziesiętnych, więc `0.1 + 0.2` nie jest równe `0.3` . Jest to jeden z najczęściej zgłaszanych błędów w JavaScript. Wynika to z przyjęcia standardu IEEE 754 (IEEE Standard for Binary Floating-Point Arithmetic).

Liczba całkowita jest najprostszym przykładem liczby.

```
let n = 3;
console.log(typeof n);
```

Liczby zmiennoprzecinkowe (ułamki)

```
let n = 2.34;
console.log(typeof n);
```

Liczby ósemkowe

Liczba zaczynająca się od `0` uznawana będzie za liczbę ósemkową, np. `0377` odpowiada dziesiętnej liczbie `255`.

```
let n = 0377;
console.log(typeof n);
```

Liczby szesnastkowe

Liczba zaczynająca się od `0x`

```
let n = 0x00;
console.log(typeof n);
console.log(n);

let n1 = 0x00;
console.log(typeof n1);
console.log(n1);
```

Notacja wykładnicza (wykładnik potęg)

`1e1` (również można zapisać to tak `1e+1`, `1E1`, `1E+1`) odpowiada liczbie `1` z jednym zerem, czyli `10`. Analogicznie `2e+3` oznacza `2` z trzema zerami czyli `2000`.

```
console.log(1e+1);
console.log(2e-3);
```

Nieskończoność (Infinity)

`Infinity` jest to specjalna wartość w języku JavaScript. Jest liczbą co łatwo sprawdzić za pomocą operatora `typeof` . Największa liczba jaką możesz użyć w JavaScript, to `1.7976931348623157e+308`, a najmniejsza to `5e-324`

```
console.log(1e+309);
console.log(typeof Infinity);
console.log(10 / 0);
```

Not a Number (NaN)

`NaN` jest specjalną wartością, która też jest liczbą

```
console.log(typeof NaN);
console.log(10 * 'f');
```

Łańcuchy znaków (string)

Typ `string` jest nietypowy w tym sensie, że jest jedynym typem prostym, który nie ma określonego rozmiaru. Może on przechowywać zero lub więcej znaków Unicode reprezentowanych przez 16-bitowe liczby całkowite. (Kiedy powstawał JavaScript Unicode opierał się wyłącznie na 16-bitowych znakach)

Każdemu znakowi nadawany jest numer w ciągu. Pierwszy znak ma pozycję zero, drugi 1 itd. Czyli pozycja ostatniego znaku to długość ciągu minus jeden.

	C	O	D	E	M	E
pozycja	0	1	2	3	4	5

Ciągi znakowe są definiowane w cudzysłowie (") lub pomiędzy apostrofami ('). Odwrotny ukośnik \ w zapisie literału ma specjalną funkcję, gdyż zmienia znaczenie znaku po nim występującego. Pozwala to na zapis tych znaków, które nie mogą występować w literałach - np. cudzysłowy lub znaki kontrolne.

```
console.log(typeof 'codeme');
console.log(typeof '1');
console.log(typeof 1);
```

znaki specjalne (przed nimi użyj \ ):

",
',
\,
/,
b - znak cofnięcia,
f - znak wysunięcia strony,
n - znak nowej lini,
r - znak powrotu karetki,
t - znak tabulacji,
u - 4 cyfry heksodecymalne

Łańcuchy znakowe są niezmienne. Raz utworzone, nigdy nie mogą być zmienione. Można za to tworzyć nowe za pomocą operatora +

Łączenie łańcuchów znaków (konkatenacja) odbywa się za pomocą operatora +

```
let s1 = 'raz';
let s2 = 'dwa';
let s = s1 + s2;
console.log(s);
```

Konwersja łańcuchów

Jeżeli jako argument działania artretycznego zostanie przekazany łańcuch, zostanie on zamieniony na liczbę.
Dotyczy to wszystkich działań z wyjątkiem dodawania (konkatenacja)

```
let s = '1';
s = 3 * s;

console.log(typeof s);
```

null

Typ null ma tylko jedną wartość `null` . Oznacza nic. Niektórzy twierdzą, że służy do reprezentowania obiektu, który nie istnieje - o czym ma świadczyć fakt, że `typeof null` zwraca `'object'` . TC39, czyli komitet odpowiedzialny za zaprojektowanie i utrzymywanie JavaScriptu, przyznał, że to błąd, ale jeśli przyjmiemy, że `null` jest pustym wskaźnikiem do obiektu, rezultat `'object'` można uznać za uzasadniony.

undefined

Typ `undefined` (nieokreślony) ma tylko jedną wartość: `undefined` . Kiedy deklarowana jest zmienna, ale nie jest inicjalizowana (nie przypisuje się jej wartości), to domyślnie otrzymuje ona wartość `undefined`

Zmienna o wartości `undefined` to nie to samo co zmienna, która nie została zadeklarowana, ale operator `typeof` tego nie rozróżnia.

Różnice może są nieduże pomiędzy `undefined` i `null` jednak mogą mieć ogromne znaczenie podczas działań artmetycznych.

```
console.log(1 + undefined);
console.log(1 + null);
```

Funkcje

Najmocniejszą stroną JavaScript jest implementacja funkcji, tu zrobiono prawie wszystko dobrze. Funkcje są sercem JavaScript. Funkcje to zbiór instrukcji, są podstawową jednostką modularną języka JavaScript. Umożliwiają ponowne użycie kodu, ukrywanie informacji oraz stosowanie kompozycji. Funkcje z zasady służą do określenia zachowania obiektów. Dzięki funkcjom JavaScript jest elastyczny i ekspresywny.

Funkcje są deklarowane słowem kluczowym `function`, po którym następuje (opcjonalnie) nazwa tej funkcji, zestaw argumentów (opcjonalnie), a następnie kod, jaki ma być wykonany, ujęty w nawiasy klamrowe - ciało funkcji.

```
function nazwaFunkcji(arg0, arg1, .., argN) {
    // ciało funkcji - instrukcje
}
```

Funkcja składa się z:

- Słowo kluczowe `function`
- Nazwa funkcji - *opcjonalnie*
- Oczekiwane parametry (argumenty) - *opcjonalnie* . Funkcja może mieć ich zero lub więcej. Jeżeli jest ich więcej niż jeden, parametry rozdziela się przecinkiem.
- Blok kodu - ciało funkcji.
- Instrukcja `return`, która pozwala na zwrócenie obliczonej wartości funkcji. Funkcja zawsze zwraca wartość, jeżeli nie robi tego w sposób jawny albo używa instrukcji `return` bez argumentu to jako swoją wartość funkcja zwraca `undefined`.

```
function suma(a, b) {
    let c = a + b;

    return c;
}

// wywołanie funkcji

suma(5, 7);
```

Kiedy definiuje się funkcję można określić oczekiwane parametry. Funkcja nie musi pobierać parametrów, ale jeżeli oczekuje, że je otrzyma, a programista podczas jej wywoływania ich nie poda, JavaScript automatycznie przypisze im wartość `undefined`.

```
suma(1); // wyświetli NaN bo 1 + undefined
```

Jeżeli funkcja otrzyma więcej parametrów niż oczekuje - nadwyżka zostanie zignorowana.

```
suma(1, 10, 3, 5, 7); // wyświetli 11
```

Dodatkowym parametrem dostępnym funkcjom podczas ich wywołania jest (pseudo)tablica `arguments`. Daje ona funkcji dostęp do wszystkich argumentów, które zostały przekazane podczas wywołania, również te nadmierne. Umożliwia to pisanie funkcji pobierających nieokreślone liczby parametrów.

```
function sumaNaDopalaczach() {
    const ln = arguments.length;
    let wynik = 0;

    for (let i = 0; i < ln; i += 1) {
        wynik += arguments[i];
    }

    return wynik;
}
```

Z powodu błędu projektowego, obiekt `arguments` nie jest prawdziwą tablicą, ale obiektem udającym tablicę (pseudo tablicą). Posiada on własność length, ale brakuje mu wszystkich metod jakie mają tablice.

Funkcje ECMAScript nie mogą być przeładowywane (przeciążane - overloading). Zdefiniowanie dwóch funkcji o tej samej nazwie w tym samym zakresie nie spowoduje błędu, ale używana będzie tylko ostatnia ze zdefiniowanych funkcji.

Dla programistów, którzy używali przeciążeń w innych językach wysokopoziomowych może to się wydawać dziwne i irytujące, ale można obejść to ograniczenie przy użyciu obiektu `arguments`.

Łańcuch zakresów

```
let a = 1;

function fn() {
     let b = 2;

     function innerFn() {
          let c = 3;

          console.log(a, b, c);
     }

     return innerFn;
}
```

Funkcja wewnętrzna ma dostęp do parametrów i zmiennych funkcji zewnętrznej z wyjątkiem `this` i `arguments`.

Zasięg leksykalny

Oznacza to, że funkcje tworzą własne środowisko (zakres) podczas definicji, a nie podczas wywołania

```
let z = 100;

function fnA() {
     let z = 1;

     fnB();
}

function fnA() {
     return z;
}

fnA();
```

`fnA()` i `fnB()` nie współdzielą zakresów lokalnych. Podczas definicji funkcja zapamiętuje swoje środowisko - łańcuch zakresów. Nie oznacza to, że funkcja pamięta każdą konkretną zmienną. Zmienne można dodawać, usuwać, aktualizować, a funkcja zawsze będzie widziała aktualny stan zmiennej. `fnB()` nie została zdefiniowana. `fnA()` posiada tylko wiedzę o swoim zakresie, aby wszystko było automatycznie dostępne.

Można usuwać `fnB()` i dodawać z innym ciałem, a `fnA()` nadal będzie działać - musi znać sposób dostępu do swojego zakresu.

Funkcje predefiniowane

Istnieje kilka funkcji, które zostały wbudowane w język JavaScript. Poniżej ich lista:

```
parseInt()
parseFloat()
isNaN()
isFinite()
encodeURI()
decodeURI()
encodeURIComponent()
decodeURIComponent()
eval()
```

funkcje takie jak `alert()`, `confirm()`, `prompt()` są funkcjami środowiska przeglądarki.

Instrukcje

Instrukcje są zwykle wywoływane z góry do dołu. Kolejność wykonania może być zmieniona poprzez zastosowanie instrukcji warunkowych `if` oraz `switch`, instrukcji pętli `while`, `for`, `do`, instrukcji przerywających `break`, `return`, `throw` i przez wywołanie funkcji.

Blok jest zbiorem instrukcji ograniczonych nawiasami klamrowymi. W przeciwieństwie do wielu języków blok w JavaScript **nie tworzy nowego zasięgu**, dlatego kiedy korzystamy z deklaracji zmiennych poprzez operator `var` nie ma sensu tworzenia (deklarowania) zmiennych wewnątrz bloku. Zmiany wprowadzone w ECMA6 polegające na dodaniu operatorów `let` i `const` zmieniają dotychczasowe spojrzenie na zasięg zmiennych. Operatory `let` i `const` pozwalają zdefiniować zmienne które mają zasięg blokowy.

Instrukcja if

```
if (/* wyrażenie warunkowe */) {
     // then
     // … kod …
} else {
     /*
          opcjonalny blok kodu
          wykonywany kiedy
          wyrażenie warunkowe
          nie jest spełnione
     */
}
```

Instrukcja `if` zmienia przepływ wykonywania programu na podstawie wartości wyrażenia warunkowego. Blok `then` jest wykonywany jeżeli `wyrażenie warunkowe` jest prawdziwe, w przeciwnym wypadku wykonywany jest `opcjonalny blok.`

Jak widać na przykładzie i wynika to z opisu możemy wyróżnić trzy podstawowe rzeczy składające się na instrukcję if

- instrukcja if
- warunek w nawiasie
- blok kodu, który ma być wykonany , jeśli warunek jest spełniony - podawany opcjonalnie

Wartości, które **nie są** uznawane za prawdziwe: `false`

- `null`
- `undefined`
- pusty łańcuch znaków ''
- liczba `0`
- liczba `NaN`

Alternatywna składnia if else

```
let mojaZmienna = /* wyrażenie warunkowe */ ?

    /* then */ :

    /* opcjonalny blok kodu */;
```

Instrukcja switch

Kiedy warunki zawierają za dużo części `else if` warto rozważyć zamianę instrukcji `if` na instrukcję `switch`

```
function insertElement(target, position, element) {
    switch (position.toLowerCase()) {
    case 'beforebegin':
        target.parentNode.insertBefore(
            element,
            target
        );
        break;
    case 'afterbegin':
        target.insertBefore(
            element,
            target.firstChild
        );
        break;
    case 'beforeend':
        target.appendChild(element);
        break;
    case 'afterend':
        target.parentNode.insertBefore(
            element,
            target.nextSibling
        );
        break;
    }
```

Instrukcja switch składa się z:

- instrukcja `switch`
- wyrażenia w nawiasie
- bloków `case` otoczonych nawiasami klamrowymi
- po instrukcji `case` występuje wyrażenie, którego wartość jest porównywana z wyrażeniem instrukcji `switch`
- blok `case` powinien być zakończony instrukcją przerwania, zazwyczaj `break`. Nie jest to obowiązkowe, ale po wykonaniu kodu związanego z instrukcją `case` zostanie wykonany blok następny w kolejce - co zazwyczaj nie jest działaniem pożądanym.
- instrukcja `default` jest opcjonalna. kod zostanie wykonany jeżeli wartość wyrażenia instrukcji `switch` nie będzie pasowała do żadnej wartości instrukcji `case`

Instrukcja while

Instrukcja `while` wykonuje prostą pętle. Dopóki wyrażenie jest prawdziwe, blok jest wykonywany, w przeciwnym wypadku pętla się kończy.

```
let i = 10;

while (i -= 1) {
     console.log(i);
}
```

Instrukcja do .. while

Instrukcja `do .. while` jest podobna do instrukcji `while`. Jedyną różnicą jest to, że warunek jest testowany po wykonaniu bloku, a nie przed. Oznacza to, że w tej instrukcji blok zawsze będzie wykonany przynajmniej raz.

```
let i = 10;

do {
     console.log(i);
} while (i -= 1);
```

Instrukcja for

W instrukcji `for` kontrola należy do trzech opcjonalnych deklaracji: inicjalizacji, warunku i inkrementacji. Pierwsza wykonywana jest inicjalizacja, która zazwyczaj uruchamia licznik pętli. Następnie sprawdzany jest warunek - zazwyczaj polega on na porównaniu licznika z zadaną wartościa oznaczającą koniec pętli. Jeżeli warunek został pominięty przypisywana jest mu wartość prawdziwa. Jeśli warunek nie jest prawdziwy, pętla jest przerywana, w przeciwnym wypadku wykonywany jest blok z kodem, po nim wywoływana jest inkrementacja i ponownie pętla zaczyna działanie od sprawdzenia warunku.

```
for (let i = 0; i < 10; i += 1) {
     console.log(i);
}
```

```
for (let i = 10; i -= 1;) {
     console.log(i);
}
```

```
let i = 10;

for (; i -= 1;) {
     console.log(i);
}
```

```
let i = 0;

for (;;) {
     console.log(i);
     i += 1;
     if (i === 10) {
          break;
     }
}
```

Obiekty

Proste typy w JavaScript to liczby (`number`), łańcuchy znaków (`string`), typy logiczne (`true` i `false`), `null` i `undefuned`. Wszystkie pozostałe wartości to obiekty.

Liczby, łańcuchy znaków i wartości logiczne są podobne do obiektów w tym sensie, że mają metody - są one jednak niezmienne (posiadają obiekty obudowujące).

Obiekty w JavaScript są modyfikowalnymi kolekcjami asocjacyjnymi. Tablice są obiektami, funkcje są obiektami, wyrażenia regularne są obiektami i obiekty są obiektami ;).

Obiekty to kolekcje właściwości, gdzie każda właściwość ma nazwę i wartość. Nazwą właściwości może być dowolny łańcuch, również pusty. Jeśli nazwa właściwości jest poprawną nazwą w JavaScript (zaczyna się od litery i zawiera litery, cyfry i podkreślnik) i nie jest słowem zastrzeżonym, to nie musi być pisana w cudzysłowie. Do oddzielenia wyliczanych właściwości używa się przecinków.

Obiekty w JavaScript są bezklasowe. Są wygodne do przechowywania i organizacji danych. Obiekty mogą zawierać inne obiekty, więc nadają się do reprezentacji struktury drzewa i grafów.

Literały obiektowe są bardzo wygodną formą zapisu wartości nowych obiektów. Literał obiektowy jest parą nawiasów klamrowych otaczających zero lub więcej par `nazwa: wartość`.

```
const obj = {
     pole: 1,
     'ale o co chodzi?': 'nie wiem',
     '!@#$%^&*': true
};
```

Obiekty różnią się od tablic m.in. tym, że programista samodzielnie definiuje klucze. W niektórych językach programowania istnieją rozróżnienia na tablice indeksowane - których kluczami są liczby i tablice asocjacyjne - których kluczami są łańcuchy znaków. W języku JavaScript tablicom indeksowanym odpowiadają tablice (`Array`) często nazywane także listą, a tablicom asocjacyjnym - obiekty (`Object`).

W programowaniu mówi się, że tablice zawierają elementy, a obiekty, mają pola. Dla JavaScript to rozróżnienie nie ma znaczenia.

Pola obiektu mogą zawierać funkcję, bo one są danymi. Takie pola nazywa się metodami. Wartość właściwości może być pobrana z dowolnego wyrażenia, włączając inny literał obiektowy.

Obiekty mogą być zagnieżdżone:

```
const samochod = {
     marka: 'Audi',
     model: 'A4',
     wyposazenie: {
          poduszki: 6,
          'elektryczne szyby': true
     }
};
```

Wartości mogą być pobierane z dowolnego obiektu przy użyciu nawiasów kwadratowych, np. `mojObiekt['!@#$%^&*']`. Jeżeli nazwa własności jest poprawną nazwą JavaScript i nie jest słowem zarezerwowanym (do ES5 - w ES5 można już tworzyć własności z nazw zarezerwowanych) można użyć notacji z kropką zamiast nawiasów kwadratowych, np. `mojObiekt.pole` - zalecane.

Próba dostępu do nieistniejącego pola kończy się zwróceniem wartości `undefined`.

Notacji nawiasowej musimy użyć także w sytuacji, kiedy nazwa pola, do którego chcemy sięgnąć, nie jest znana w czasie pisania kodu, przypisujemy jej wartość zmiennej.

```
const NAME = 'to jest nazwa wlasnosci';

console.log(myObject[NAME]);
```

Wartość właściwości obiektu może być zdefiniowana poprzez przypisanie. Jeżeli właściwość o danej nazwie istnieje już w obiekcie, jej wartość zostanie nadpisana.

`this` - zawsze wskazuje na obiekt, który wywołuje daną metodę. Czyli z wnętrza metody poprzez `this` odwołujemy się do obiektu do którego należy metoda.

```
const samochod = {
     marka: 'Audi',
     model: 'A4',
     wypiszMarkaModel: function () {
          console.log(this.marka + ' ' + this.model);
     }
};
```

W tym kontekście `this` wskazuje na obiekt `samochod`.

Wzorzec funkcji fabrykującej

```
function auto(marka, model) {
    function wypisz() {
        console.log('Marka: ' + this.marka);
        console.log('Model: ' + this.model);
    }
    return {
        marka: marka,
        model: model,
        wypisz: wypisz
    };
}
```

Polega na utworzeniu funkcji zwracającej obiekt. Dzięki takiemu rozwiązaniu zanim uzyskamy oczekiwany obiekt może być przetworzonych wiele danych np. podanych w parametrach wywołania funkcji. W przykładzie widać, że przy tej metodzie tworzenia egzemplarza obiektu, za każdym razem, tworzona jest funkcja (w przykładzie - `wypisz`) i w efekcie każdy obiekt ma własną wersję tej funkcji.

Wzorzec konstruktora

Obiekty można tworzyć przy użyciu funkcji zwanych konstruktorami. Konwencja nakazuje tworzyć konstruktory zaczynając od **wielkiej litery** co odróżni je od funkcji.

```
function Auto(marka, model) {
    this.marka = marka;
    this.model = model;
}
```

konstruktor jest wywoływane przy pomocy operatora `new`

```
const auto = new Auto('Audi', 'A4');
console.log(auto.marka);
```

Wywołanie konstruktora bez operatora `new` nie spowoduje błędu lecz prowadzi do nieoczekiwanych wyników.

```
const cosik = Auto('Cosik', 'Ktosik');
console.log(cosik);
```

Przy takim wywołaniu nie powstaje nowy obiekt, a funkcja `Auto()` zwróci domyślną wartość - `undefined`. Operator `this` wewnątrz `Auto()` odnosi się do obiektu globalnego środowiska wywołania (w przeglądarce `window`) . Jeżeli konstruktor zostanie wywołany poprawnie (z operatorem `new`) zostanie utworzony nowy obiekt do którego odnosić się będzie `this`.

Obiekty wbudowane

Dokument ECMA-262 definiuje obiekty wbudowane jako obiekty udostępnione przez implementacje ECMAScript niezależnie od środowiska, w którym działa. Krócej : obiekty własne to referencje zdefiniowane w dokumencie ECMA-262.

W języku ECMAScript, w zasadzie nie występują klasy w tradycyjnym tego słowa znaczeniu. ECMAScript definiuje „definicję obiektów", które są logicznie tożsame klasom w innych językach programowania.

Object

Jest to obiekt bazowy, którego potomkami są wszystkie inne obiekty w języku JavaScript - wszystkie inne obiekty z niego dziedziczą.

Aby utworzyć obiekt możesz skorzystać z notacji literału (zalecana) albo z konstruktora `Object`

```
const obj1 = {};
const obj2 = new Object();
```

Spis i opis podstawowych pól i metod tego typu referencyjnego `Object` znajdziesz np. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

Function

Funkcje w JavaScript są obiektami. Mają swój własny konstruktor `Function`, który pozwala tworzyć funkcje w sposób, który nie jest zalecany, wręcz niewskazany i zarazem uważany za niebezpieczny, gdyż String jest wykonywalnym kodem ;)

Istnieją trzy równoważne sposoby definiowania funkcji:

```
function suma(a, b) {
   return a + b;
}

var suma = function (a, b) {
   return a + b;
};

var suma = new Function('a', 'b', 'return a + b');
```

Konstruktor `Function` przyjmuje dowolną liczbę parametrów, przy czym ostatni parametr jest traktowany jako ciało funkcji.

Wujek dobra rada podpowiada: Należy unikać wszelkich funkcji, które jako argument pobierają kod w postaci łańcucha znaków - np. `eval` czy też definiowanie funkcji przez `new Function`

Opis pól i metod każdej funkcji, którą utworzymy (jak już kilka razy podkreślałem funkcje są obiektami więc mają swoje pola i metody) znajduje się m.in. pod adresem https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function

prototype

Ważnym polem funkcji jest pole `prototype`.

Funkcje w JavaScript są obiektami dlatego dziedziczą z typu referencyjnego `Object` pole `prototype`. Pole to jest tworzone w momencie definicji funkcji.

Pole `prototype` jest obiektem, który można rozszerzyć o pola i metody.

```
(function () {
    function fn(a, b) {
        return a + b;
    }
    console.log(fn.length);
    console.log(fn.constructor);
    console.log(typeof fn.prototype);
}());
```

Rozszerzenia obiektu `prototype` nie mają wpływu na funkcję, natomiast zostaną one użyte tylko podczas wywołania funkcji jako konstruktora (czyli z operatorem `new`).

Dodawanie pól i metod przy użyciu prototype

```
function Auto(marka, model) {
    this.marka = marka;
    this.model = model;

    this.wypiszMarkaModel = function () {
        console.log(this.marka + ' ' + this.model);
    }
}
```

Konstruktor `Auto()` poprzez `this` dodaje dwa pola i metodę. Innym sposobem rozszerzenia nowego obiektu jest dodwanie pól i metod do obiektu `prototytpe` konstruktora.

```
function Auto(marka, model) {
    this.marka = marka;
    this.model = model;
}

Auto.prototype.wypiszMarkaModel = function () {
    console.log(this.marka + ' ' + this.model);
}
```

```
const auto = new Auto('Audi', 'A4');
auto.wypiszMarkaModel();
```

Pola i metody dodane do `prototype` stają się dostępne zaraz po utworzeniu nowego obiektu poprzez dany konstruktor.

Obiekty są przekazywane przez referencję, dlatego też zmiana `prototype`, pociąga za sobą zmiany we wszystkich dziedziczących z niego obiektach, nawet w tych utworzonych wcześniej.

```
Auto.prototype.ustawModel = function (model) {
    if (model) {
        this.model = model;
    }
}

auto.ustawModel('A5');
```

Metody rozszerzające typ referencyjny Function

metoda `.apply()` - pobiera dwa argumenty: obiekt, który ma wskazywać na `this` i tablicę argumentów do przesłania do wywoływanej funkcji.

```
function wypiszKolor(prefix, suffix) {
    console.log(prefix + this.color + suffix);
}

const auto = {
    marka: 'Audi',
    color: 'granatowy'
};

wypiszKolor.apply(auto, [
     'kolor:',
     ' to ładny kolor'
]);
```

metoda `.call()` - działa podobnie jak metoda `.apply()` tyle, że zamiast tablicy argumentów podajemy kolejne argumenty po przecinku

```
wypiszKolor.call(auto, 'kolor: ', ' to ładny kolor');
```

metoda `.bind()` - tworzy nową funkcję, która po wywołaniu jako this ma ustawioną referencję do obiektu podanego jako pierwszy parametr. Metoda ta opcjonalnie przyjmuje jeszcze sekwencje atrybutów, które będą przekazane podczas wyłania nowej funkcji jako parametry tej funkcji. Więcej informacji oraz przykładów:

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind

```
const kolor = {
    color: 'czerwony',
    wypiszKolor: function (prefix, suffix) {
        console.log(prefix + this.color + suffix);
    }
};

const auto = {
    marka: 'Audi',
    color: 'granatowy'
};

const autoWypiszKolor = kolor.wypiszKolor.bind(auto);
autoWypiszKolor();
```

Array

Konstruktor `Array()` tworzy tablicę, zalecane jest jednak tworzenie tablic poprzez literał tablicowy `[]`.

Poniższe deklaracje są równoważne.

```
const arr1 = ['a', 'b', 'c'];

const arr2 = new Array('a', 'b', 'c');
```

Wywołanie `Array` jako funkcji, a nie konstruktora, nie spowoduje błędu ponieważ środowisko samo zamieni takie wywołanie na wywołanie konstruktora czyli z `new` jednak takie pisanie kodu może wprowadzić chaos i spowodować nieczytelność kodu. Dlatego też zalecane jest stosowanie literałów do tworzenia tablic.

Tablice są obiektami posiadającymi wyjątkowe cechy:

- Pola nazywane są automatycznie za pomocą liczb od zero w górę
- Posiadają pole `.length`, które zawiera liczbę elementów tablicy
- Poza metodami odziedziczonymi z `Object` posiadają własne metody wbudowane.

Wykaz i opis rozszerzeń (metod) typu referencyjnego `Array` :

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

String

Konstruktor `String()` służy do tworzenia obiektowych reprezentacji wartości prostych typu *`string`*. Zalecane jest jednak korzystanie z prostego typu danych.

```
let str = 'Jo man';
console.log(typeof str);

let strObj = new String('oto i stringus');
console.log(typeof strObj);
```

Obiekt String ma właściwość - `.length`, która zwraca liczbę znaków w ciągu.

```
console.log(str.length);
console.log(strObj.length);
```

Obiekt String ma sporą ilość własnych metod, których opis i listę możesz zobaczyć tutaj: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Sam łańcuch znaków nie jest obiektem i nie posiada żadnych pól i metod. Język JavaScript pozwala na traktowanie prostych łańcuchów jak obiekty.

Funkcja `String()` użyta bez operatora `new` zamieni parametr na typ prosty. Jeżeli wartość parametru będzie obiektem to zostanie wywołana metoda `.toString()`

```
console.log(String(997));
console.log(String({n: 20}));
console.log(String([1, 2, 3]));
```

Boolean

Obiekty utworzone za pomocą konstruktora `Boolean` nie są specjalnie przydatne, a czasami mogą prowadzić do powstania błędów logicznych.

```
var log = new Boolean();
```

Zmiennej log został przypisany nowy obiekt a nie wartość typu *`boolean`*.

Inaczej przedstawia się sprawa z `Boolean()` wywołanym jako zwykła funkcja (bez użycia operatora `new`), a nie konstruktor.

W ten sposób można zmieniać na typ logiczny wartość należącą do innego typu danych - odpowiada to zastosowaniu podwójnej negacji - tz. `!!wartosc`

```
console.log(Boolean('stringus'));
console.log(Boolean(''));
console.log(Boolean({}));
```

Wujek dobra rada podpowiada: Warto mieć świadomość, że możliwe jest tworzenie obiektów `Boolean`, ale w praktyce najlepiej korzystać z prostych wartości logicznych (`true` i `fales`)

Number

`Number()` można użyć w dwojaki sposób:

- jako konstruktora, za pomocą którego tworzy się nowy obiekt.
- jak normalnej funkcji zmieniającej dowolną wartość na liczbę (podobnie jak `parseInt()` i `parseFloat()`)

```
let num1 = new Number('13.10');
console.log(typeof num1, num1);

var num2 = Number('13.10');
console.log(typeof n2, num2);
```

Opis konstruktora `Number`, jego własności i metod można znaleść tutaj:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

Math

`Math` to pojedynczy obiekt (singleton) posiadający własności, które mogą być funkcjami - metodami. Pozwala wykonywać operacje matematyczne.

Nie można zmieniać wartości pól i metod obiektu `Math`.

Wykaz pól i metod obiektu `Math` znajduje się tutaj:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Inne... - zobacz
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Dziedziczenie

Obiektowe języki programowania muszą obsługiwać dziedziczenie, czyli możliwość korzystania z metod i właściwości jednej klasy przez inną klasę.

Klasycznym modelem dziedziczenia są figury geometryczne. Istnieją dwa typy figur geometrycznych - elipsy i wielokąty. Koła to rodzaj elipsy z jednym ogniskiem. Trójkąty, czworokąty ..., to rodzaje wielokątów z różną ilością boków. Kwadrat to rodzaj czworokąta z wszystkimi bokami równymi.

Przykład ten definiuje obiekt bazowy "Figura" dla obiektów "Elipsa" i "Wielokat". "Elipsa" ma jedną właściwość "ogniska". "Kolo" jest potomkiem "Elipsa". ...

Mechanizm prototypowania jest zaciemniany przez niektóre skomplikowane elementy składni, które wyglądają bardziej klasycznie. Zamiast pozwolić obiektom dziedziczyć bezpośrednio z innych obiektów JavaScript wprowadza niepotrzebny poziom abstrakcji, w którym obiekty tworzone są przy pomocy funkcji konstruktorów.

Sposoby dziedziczenia

W ECMAScript można implementować dziedziczenie na kilka sposobów. Wynika to z faktu, że dziedziczenie w JavaScript nie jest jawne, tylko emulowane. Oznacza to, że interpreter nie obsługuje wszystkich szczegółów związanych z dziedziczeniem. Zadaniem programisty jest obsługa dziedziczenia w sposób najbardziej adekwatny do okoliczności.

Wiązanie łańcuchowe prototypów - dziedziczenie pseudoklasyczne + maskowanie obiektu

```
function KonstruktorA() {
}
Object.assign(KonstruktorA.prototype, {
    kolor: 'niebieski',
    powiedzKolor: function () {
        console.log(this.kolor);
    }
});

function KonstruktorB() {
}
KonstruktorB.prototype = new KonstruktorA();
```

Obiekt `KonstruktorB.prototype` staje się egzemplarze `KonstruktorA`

Ma to swoje wady ponieważ własności i metody własne konstruktora `KonstruktorA` zostaną przypisane do obiektu `prototype`. Druga rzecz to to, że `KonstruktorA`, musi mieć możliwość stworzenia egzemplarza obiektu bez podania parametrów podczas wywołania, jeżeli takowe przyjmuje. W rozwiązaniu problemów wyżej opisanych pomoże maskowanie obiektu oraz zastosowanie dziedziczenia prototypowego obiektu `prototype`:

```
function A(kolor) {
    this.kolor = kolor;
}
A.prototype.powiedzKolor = function () {
   console.log(this.kolor);
}

function B(kolor, nazwa) {
    A.call(this, kolor);
    this.nazwa = nazwa;
}

B.prototype = Object.create(A.prototype);

B.prototype.powiedzNazwa = function () {
    console.log(this.nazwa);
}

var obA = new A('czerwony');
var obB = new B('zielony', 'Kosmita');

obA.powiedzKolor();
obB.powiedzKolor();
obB.powiedzNazwa();
```

Czasami zdarza się, że konstruktor musi przyjąć bardzo dużo parametrów. Może to być kłopotliwe zwłaszcza zapamiętanie ich kolejności. W takich przypadkach lepszym rozwiązaniem może być napisanie konstruktora, który pobiera pojedyncze określenie obiektu - ten wzorzec warto stosować również podczas definiowania pojedynczych funkcji.

```
const obj = new Konstruktor({
    arg1: 'f',
    arg2: 's',
    ..
    argN: 'z'
});
```

Dziedziczenie prototypowe

W przypadku podejścia czysto prototypowego nie używa się konstruktorów. Skupiamy się wyłącznie na obiektach. Obiekt może dziedziczyć właściwości starego obiektu. Punktem wyjścia, w tym sposobie dziedziczenia, jest utworzenie jakiegoś pożytecznego obiektu. W następnym kroku możemy utworzyć wiele obiektów podobnych do niego.

```
const zwierz = {
     imie: 'moj ssak',
     pobierzImie: function () {
        return this.imie;
     },
     powiedz: function () {
        return this.slowka || '';
     }
};

const kot = Object.create(zwierz);
Object.assign(kot, {
    imie: 'Mruczek',
    slowka: 'miau miau',
    fn: function () {
        console.log('pije mleko');
    }
});
```