Podstawy Javascript

April 27, 2020

JavaScript. Tworzenie nowoczesnych aplikacji webowych. Tomasz Sochacki

Podstawy ECMAScript

Podstawowe elementy składni

Nawiasy

Komentarze

Deklarowanie zmiennych i stałych

Deklaruje się je przy użyciu słów let lub const. Ich nazwy nie mogą się rozpoczynać od cyfry, nie mogą zawierać białych znaków (czyli np. spacji) i nie mogą stanowić tzw. słowa kluczowego JavaScript. Do ich tworzenia wykorzystuje się notację camelCase, gdzie każde słowo (poza pierwszym) piszemy z wielkiej litery.

const userName = 'Jan';

Nazwy klas zapisujemy w notacji PascalCas e , czyli każdy wyraz, łącznie z pierwszym, zaczyna się wielką literą. W wielu językach programowania istnieje również niepisana zasada, że nazwy stałych piszemy zawsze tylko wielkimi literami, a nazwy zmiennych w odpowiedniej notacji (np. camelCas e ). W świecie JavaScript zasada ta nie jest tak rygorystycznie przestrzegana i zależy w zasadzie od ustaleń zespołu, który pracuje nad danym projektem.

const PI = Math.PI;

Tworząc nazwę zmiennej czy stałej, warto również zwracać uwagę na jej czytelność. Nie należy także bać się dłuższych nazw, jeśli faktycznie dają one czytelną informację o przeznaczeniu. Nazwy powinne by opisowe.

Typy zmiennych

W języku JavaScript wyróżniamy zmienne typu prostego i złożonego. Typy proste to undefined , null , boolean, string, number. Do typów prostych zalicza się również Symbol. Typ złożony to obiekt, ale w JavaScript nie występuje oddzielny typ dla tablic, jak w innych językach programowania. Tutaj tablica ( array ) jest po prostu obiektem, posiadającym pewne, specyficzne dla siebie właściwości.

JavaScript jest językiem z tzw. słabym typowaniem. Oznacza to, że zmienna może przechowywać wartości różnych typów i zmieniać je w trakcie działania aplikacji.

let value = 'Jan';

typeof value; // 'string'

value = 5;

typeof value; // 'number'

Jak widać, da się zmienić nie tylko wartość, ale i typ zmiennej. Należy jednak uważać i raczej wystrzegać się, przypisywania do zmiennej wartości różnych typów gdyż mogą one prowadzić do trudnych do zlokalizowania błędów.

Ciągi znakowe przechowywane są jako wartości typu string , a liczby jako number. W języku JavaScript jest tylko jeden typ dla liczb, który pozwala przechowywać liczby w różnych zapisach zarówno jako liczby całkowite, jak i zmiennoprzecinkowe. Istnieje jeszcze jeden, nowy typ bigint.

Wszystkie poniższe zapisy są poprawne i przechowują wartości typu numerycznego:

const num = 5;

const num = 5.0;

const num = .2;

const num = 2e5;

const num = -0.15;

Ponadto możliwe jest również przechowywanie liczb w zapisie szesnastkowym, ósemkowym czy binarnym.

Typ undefined oznacza dokładnie brak wartości. Może ona być przypisana bezpośrednio lub w sposób niejawny:

let value1; //niejawnie przypisane undefined

let value2 = undefined; //przypisanie bezpośrednie

Typ null oznacza natomiast pustą wartość, jednakże nie jest to ani pusty ciąg znakowy, ani wartość undefined. Typ undefined jest dość specyficzny dla JavaScript, podczas gdy typ null jest powszechnie stosowany w wielu językach programowania. Zmienna przechowująca wartość null może sprawiać wrażenie, jakby przechowywała inny typ:

const value = null;

typeof value; //'object'

Jest to spowodowane błędem, występującym już od pierwszych wersji JavaScript, który prawdopodobnie nie zostanie poprawiony ze względu na konieczność zachowania zgodności wstecznej. Typ null jest jednak zaliczany do typów prostych (ang. primitive types ).

Czasami można spotkać się również z zapisem void 0 , co jest w zasadzie tożsame z undefined.

undefined === void 0; //true

Następnym typem prostym jest boolean, który oznacza wartość logiczną prawda/fałsz (true / false). Typ ten jest bardzo często stosowany m.in. do wskazywania różnych stanów istotnych dla działania aplikacji.

let isUserLogged = false;

let shouldDisplayTotalPrice = false;

Zazwyczaj nazwy zmiennych przechowujących typ boolean zaczynają się od „is” lub „should”.

Typ złożony określany jest jako object obejmuje obiekty oraz tablice, choć jest umowny podział, gdyż w praktyce tablice w JavaScript są obiektami, a nie odrębnym typem. Deklaracja tablic i obiektów zwykłych wygląda jednak nieco inaczej.

Obiekt to pewien byt, przechowujący różne pola i funkcje. Możemy stworzyć obiekt person , który będzie posiadał pola takie jak name, age, address oraz funkcje (nazywane często metodami obiektu) jak speak, sleep itp. Obiekty takie deklaruje się z użyciem nawiasów klamrowych:

const person = {

name: 'Jan',

age: 30,

address: {

city: 'Warszawa',

country: 'Polska'

},

speak() { /\*funkcja speak\*/ },

sleep() { /\*funkcja sleep\*/ }

};

Poszczególne pola i metody rozdzielamy przecinkami, przy czym przecinek za ostatnim polem jest opcjonalny (przecinek ten określany jest jako trailing comma).

Aby odwołać się do konkretnego pola w obiekcie person, należy zastosować notację z kropką lub z nawiasami kwadratowymi:

console.log(person.name); // 'Jan'

console.log(person['name']); // 'Jan'

Zapis według drugiej formy jest stosowany, gdy chcemy odwołać się do nazwy pola, która jest przechowywana w zmiennej lub, jak w tym przykładzie, w stałej:

const fieldName = 'age';

console.log(person.age); // 30

console.log(person['age']); // 30

console.log(person[fieldName]); // 30

Gdy nazwę pola wskazujemy poprzez zmienną, to nie otaczamy jej apostrofami ani cudzysłowami, gdyż zapis:

console.log(person['fieldName']); // undefined

oznaczałby próbę pobrania wartości pola o nazwie fieldName, a takie pole nie występuje w obiekcie person.

Tablice są zbiorami pewnych danych, podobnie jak obiekty, jednakże dane te są przechowywane w określonej kolejności, a każdy element tablicy posiada swój unikalny index. Deklaruje się je przy użyciu nawiasów kwadratowych:

const fruits = ['jabłko', 'wiśnia', 'gruszka'];

Tablice mogą przechowywać wartości dowolnego typu, w tym również kolejne tablice (mówi się wtedy często o tzw. tablicach wielowymiarowych), oraz mogą przechowywać elementy o różnych typach, np.:

const values = ['jabłko', 5, {age: 30}, [1, 2, 3]];

Tablica values posiada cztery elementy: typu string , number , object i kolejną tablicę z trzema elementami. Lepiej przechowywać w tablicy wartości tylko jednego typu. Jeśli potrzebujemy tablicy z różnymi typami danych, to możemy użyć zwykłego obiektu z nazwanymi polami.

Aby odwołać się do określonego elementu tablicy, należy zastosować zapis z nawiasami kwadratowymi, podając w nich index szukanego elementu (numeracja indeksów zaczyna się od zera):

console.log(values[0]); // ' jabłko '

console.log(values[2]); // { age : 30 }

console.log(values[2].age); // 30

console.log(values[3]); // [ 1,2,3 ]

console.log(values[3][1]); // 2

Zmienne vs stałe w JavaScript

Jeszcze kilka lat temu zmienne w JavaScript deklarowało się przy użyciu słowa var. Obecnie nie jest to zalecana metoda i zmienne deklaruje się za pomocą słowa let lub cons t . Słowo let służy do deklarowania zmiennych, a const przeznaczone jest dla tzw. stałych.

Jeśli zmienna zostanie zadeklarowana z użyciem słowa let to nie jest możliwe ponowne zadeklarowanie zmiennej o tej samej nazwie, znajdującej się w tym samym zakresie.

Z tego powodu nie jest zalecane korzystanie z deklaracji ze słowem var, które pozwalało na wielokrotne tworzenie nowych zmiennych o tych samych nazwach, przez co aplikacja mogła być bardziej podatna na przypadkowe błędy programisty.

Deklarując zmienną słowem let , nie musimy od razu przypisywać jej wartości początkowej, ale możemy zrobić to w dwu krokach — pierwszym jest deklaracja, a drugim inicjalizacja zmiennej (czyli właśnie nadanie jej wartości początkowej):

let value; // deklaracja zmiennej

value = 5; // inicjalizacja zmiennej

W praktyce jednak najczęściej będziemy posługiwać się deklaracjami z użyciem słowa const.

const age = 30;

age = 10; // TypeError : Assignment to constant variable .

age; // 30

Jeśli stała została zadeklarowana z wartością typu prostego, czyli string , number, undefined , null lub boolean, to nie będzie możliwości późniejszej ingerencji w tę wartość. Jeśli jednak stała przechowuje wartość typu złożonego, czyli obiekt, to jest tzw. stałą modyfikowalną.

const letters = ['a', 'b', 'c'];

// próba zmiany typu wartości na number :

letters = 5; // TypeError : Assignment to constant variable .

letters; // [" a ", " b ", " c "]

// próba modyfikacji elementów tablicy :

letters[0] = 'nowy';

letters; // [" nowy ", " b ", " c "]

// próba dodania kolejnego elementu :

letters.push('d');

letters; // [" nowy ", " b ", " c ", " d "]

O ile nie jest możliwe przypisanie do stałej letters wartości innego typu (ani bezpośrednie przypisanie nowej, innej tablicy), to możliwa jest modyfikacja tej tablicy poprzez odwoływanie się do konkretnych jej elementów.

Podobne zachowanie występuje w przypadku przypisania do stałej jakiegoś obiektu — możliwe jest wtedy modyfikowanie wartości poszczególnych pól, dodawanie nowych, usuwanie istniejących itp. Nie można jednak do stałej przypisać całkowicie nowego obiektu lub wartości innego typu.

Operatory matematyczne

Istnieje ogólnie przyjęta zasada, aby zawsze stawiać pojedyncze spacje wokół różnego rodzaju operatorów, co zwiększa czytelność kodu i ułatwia jego analizę.

Operator przypisania (znak równości = ) pozwala na przypisanie do zmiennej jakieś konkretnej wartości:

const age = 30;

const price =100;

Wewnątrz obiektu polu przypisujemy wartość za pomocą dwukropka.

const person = { age: 30 };

Późniejszy dostęp do tego pola i zmiana jego wartości odbywa się przy użyciu znaku równości: person.age = 50;

|  |  |  |
| --- | --- | --- |
| Operator | Znaczenie | Forma przypisania |
| + | Dodawanie | x = x + 5 lub x += 5 |
| - | Różnica | x = x – 5 |
| \* | Iloczyn | x = x \* 5 lub x \*= 5 |
| / | Iloraz | x = x / 5 lub x /= 5 |
| % | Modulo | x = x % 5 lub x %= 5 |
| \*\* | Potęgowanie | x = x \*\* 2 brak formy skróconej |

Operatory matematyczne należy stosować dla wartości typu number. Dla typów string czy string i number operatory inne niż dodawanie dają poprawne rezultaty:

'5' \* 5; // 25 ( jako typ number )

'10' - 2; // 8 ( jako typ number )

'5' + 5; // ' 55 ' ( jako typ string )

5 + 5; // 10 ( jako typ number )

Konkatenacja ciągów znakowych

Konkatenacja to połączenie dwóch lub więcej ciągów znakowych, czyli wartości typu string .

Przeciążenie operatora + , czyli zmiana jego pierwotnego znaczenia.

Jeśli operator + zostanie użyty na wartościach, wśród których co najmniej jedna nie jest wartością typu number, to spowoduje on niejawne przekonwertowanie wszystkich wartości do typu string i połączenie ich w jeden ciąg znakowy.

Template strings

Ciągi znakowe w JavaScript zazwyczaj tworzymy przy użyciu

apostrofów lub cudzysłowów:

const a = 'jakiś tekst';

const b = "jakiś tekst";

Trzeci sposób to wykorzystanie tzw. odwróconych apostrofów:

const c = `jakiś tekst`;

W tym przypadku silnik JavaScript dokona interpretacji wyrażenia zawartego między odwróconymi apostrofami. Ta forma zapisu pozwala na wprowadzenie dowolnego innego wyrażenia JavaScript przy użyciu zapisu: ${ wyrażenie } :

const firstName = 'Jan';

const lastName = 'Kowalski';

const name = `${firstName} ${lastName}`; // ' Jan Kowalski '

Operatory porównania

Operatory te służą do sprawdzania zależności między dwiema wartościami.

|  |  |
| --- | --- |
| Operator | Opis |
| == | Równe wartości |
| != | Nie równe wartość |
| === | Równe typy i wartości |
| !== | Różne typy i wartości |
| > | Większe niż |
| >= | Większe bądż równe |
| < | Mniejsze niż |
| <= | Mniejsze bądż równe |

'5' == 5;

// true , bo porównywane są tylko wartości , więc różnica typów ( string vs number ) nie ma znaczenia

'5' === 5; // false , bo porównywane są wartości i typy , a te drugie są różne ( string vs number)

Operatory mniejszości i większości występują tylko w jednej postaci i w razie potrzeby dokonują konwersji typów:

5 > ‘2’ // true (number vs string)

‘5’ > 4 // true ( string vs number)

Operatory logiczne

|  |  |
| --- | --- |
| Operator | Opis |
| ! | Negacja logiczna (NOT) |
| || | Suma logiczna (OR) |
| && | Iloczyn logiczny (AND) |
| I | Suma binarna |
| & | Iloczyn binarny |
| ^ | XOR alternatywa wykluczająca |

Operator XOR zwraca true tylko wtedy, gdy albo jedna, albo druga wartość jest prawdziwa, ale nie gdy obie dają true.

Instrukcja warunkowa if-else

W języku JavaScript instrukcja warunkowa if posiada prostą składnię w postaci:

if (warunek) {

// kod wykonywany gdy warunek jest spełniony

}

Dobrą praktyką, aby zawsze w instrukcjach warunkowych stosować nawiasy klamrowe, nawet gdy do wykonania jest tylko jedno polecenie.

Warunek jest to wyrażenie, które zostaje przekonwertowane do postaci boolean ( true / fals e ) i w ten sposób warunkuje rozpoczęcie wykonywania poleceń z wnętrza instrukcji if .

W niektórych przypadkach będziemy chcieli sprawdzać również dodatkowe warunki, gdy pierwszy okaże się false . Pomocna jest w tym wypadku konstrukcja if - else if :

if (price > 100) {

// kod dla ceny większej niż 100

} else if (price > 50) {

// kod dla ceny większej niż 50 , ale mniejszej lub równej 100

} else {

// kod wykonany w przypadku niespełnienia żadnego z warunków

}

Można stosować wiele instrukcji else if , jednakże w praktyce jeśli istnieje potrzeba zastosowania dwóch lub więcej warunków else if , to najprawdopodobniej lepszym wyborem będzie instrukcja switch

Instrukcja warunkowa if posiada także wersję skróconą, określaną jako operator trójargumentowy ( ternary operator ):

warunek ? wyrażenie1 : wyrażenie2

Forma ta jest często stosowana podczas deklarowania wartości dla zmiennych oraz w instrukcjach return funkcji. Jego składnia jest następująca:

warunek ? polecenie jeśli true : polecenie jeśli false;

Nie należy go traktować jako bezpośrednią alternatywę dla instrukcji if - else if , gdyż w przypadku operatora trójargumentowego możliwe jest wykonanie wyłącznie pojedynczego polecenia.

const price = 100;

const priceToPay = price > 200 ? price \* 0.8 : price;

Alternatywny zapis bez użycia operatora trójargumentowego wyglądałby następująco:

const price = 100;

let priceToPay;

if (price > 200) {

priceToPay = price \* 0.8;

} else {

priceToPay = price;

}

Widać wyraźnie, że operator trójargumentowy istotnie skraca zapis, nie powodując utraty czytelności.

Na koniec przeanalizujmy jeszcze jeden przykład użycia instrukcji if :

if (value === ('first' || 'second')) {

// instrukcje dla true

} else {

// instrukcje dla false

}

Naszym celem jest sprawdzenie, czy wartość value jest równa ciągom znakowym first lub second. W pierwszej kolejności następuje sprawdzenie alternatywy, czyli: ('first' || 'second') Pierwszy argument, po lewej stronie operatora || , zostanie przekonwertowany do true , ponieważ jest to niepusty ciąg znakowy i to właśnie on zostanie zwrócony jako wynik.

W konsekwencji więc będziemy zawsze sprawdzać:

if (value === 'first') {

// instrukcje dla true

} else {

// tutaj wejdziemy gdy value === ' second '

}

Powino być tak:

if (value === 'first' || value === 'second') {

// instrukcje dla true

} else {

// instrukcje dla false

}

Instrukcja warunkowa switch

Czasami zdarzają się sytuacje, w których sprawdzana zmienna może przyjmować kilka różnych wartości i aplikacja powinna zachować się różnie, w zależności od tego, jaka wartość zostanie rozpoznana.

switch (wartość) {

case wartość\_1:

// polecenia do wykonania w przypadku gdy wartość === wartość\_1

break ;

case wartość\_2:

// polecenia do wykonania w przypadku gdy wartość === wartość\_2

break ;

...

case wartość\_n:

// polecenie do wykonania w przypadku gdy wartość === wartość\_n

break;

default :

// polecenia do wykonania, gdy nie dopasowano żadnego z warunków

}

Operator == sprawdza wartości po ewentualnej konwersji typów i może być zastosowany w instrukcji warunkowej if. W instrukcji switch zawsze sprawdzane są wartość i typ czyli stosuje się operator === .

Pętla for

for (let i = 0; i < 5; i += 1) {

console.log(`Wartość zmiennej i wynosi: ${i}`);

}

Ogólna postać pętli for:

for (wartość\_początkowa; warunek\_zakończenia\_iteracji; modyfikacja\_wartości) {

}

W każdej iteracji wartość możemy modyfikować w dowolny sposób.

Zapisy

i++ ;

i += 1 ;

są równoważne i w obu przypadkach zmienna jest zwiększana o jeden.

W programowaniu wyróżniamy cztery podstawowe przypadki takiego zapisu:

i++ // inkrementacja (postinkrementacja)

i-- // dekrementacja (postdekrementacja)

++i // preinkrementacja

--i // predekrementacja

Instrukcja break i continue

Instrukcja break pozwala zakończyć działanie pętli for niezależnie od tego, czy spełniony jest warunek określony jako drugi parametr w definicji pętli.

Instrukcja continue z kolei pozwala na zaprzestanie wykonywania dalszych poleceń w danej iteracji i przejście do kolejnego cyklu.

for (let i = 0; i < 100; i += 1) {

// omijamy wartości niepodzielne przez 5 :

if (i % 5 !== 0) {

continue ;

}

// jeżeli i będzie większe od 20 to kończymy działanie pętli

if (i <= 20) {

console.log(`Obecna wartość: ${i}`);

} else {

break ;

}

}

Pętla for może być także wykorzystana do iterowania po elementach tablicy.

const letters = ['a', 'b', 'c'];

for (let index = 0; index < letters.length; index += 1) {

console.log(letters[index]) ;

};

Pętle for-in oraz for-of

Najczęściej konstrukcja for - in jest wykorzystywana do iteracji po nazwach pól w obiektach, a pętla for - of do iterowania po wartościach tablicy.

const person = {

firstName: 'Jan',

lastName: 'Kowalski'

};

for (const fieldName in person) {

console.log(fieldName) ;

}

// firstName

// lastName

const letters = ['a', 'b', 'c'];

for (const letter of letters) {

console.log(letter) ;

};

Nie stosuj konstrukcji for - in do iterowania po tablicach. Gdybyśmy dokonali iteracji po tablicy letters konstrukcją for - in , to w konsoli zobaczylibyśmy trzy wartości: '0', '1', '2' czyli indeksy poszczególnych elementów jako string . Tablice w JavaScript są obiektami, w których poszczególne elementy stanowią pola obiektu, a kluczami dla tych pól są właśnie indeksy.

Pętle while oraz do-while

let n = 3; while (n) { console.log(n) ; n -= 1; } // 3 // 2 // 1 // 0

do { // operacje wykonywane w pętli } while (warunek);

Konwersja typów zmiennych

JavaScript jest to język ze słabym typowaniem i często będziemy dokonywać konwersji; najczęściej będą to konwersje typów string i number do boolean.

W JavaScript do false przekonwertują się następujące wartości:

false

undefined

null

NaN (Not a Number)

'' lub "" lub `` (pusty ciąg znakowy)

0 (cyfra zero)

Ciąg zawierający znak spacji nie jest już pustym ciągiem i zostanie przekonwertowany do true.

Każdy obiekt i każda tablica w JavaScript jest traktowana jako wartość true , niezależnie od tego, czy zawiera jakieś elementy, czy nie.

Aby jawnie przekonwertować dowolną wartość do typu boolea n , można przed tą wartością postawić dwa znaki wykrzyknika.

!!""; // false

!!undefined; // false

!!0; // false

!!null; // false

!!false; // false

!![]; // true – pusta tablica

!!{}; // true – pusty obiekt

!![1,2,3]; // true

!!Boolean(true); // true - jawna konwersja

!!(new Boolean(true)); // true – stworzenie obiektu

!!Boolean(false); // false – jawna konwersja

!!(new Boolean(false)); // true – stworzenie obiektu

const valA = Boolean(false);

const valB = new Boolean(false);

typeof valA; // " boolean "

typeof valB; // " object "

const message = "jakiś tekst";

if (message) {// niejawna konwersja string na true

// ten kod się wykona

} else {

// ten kod się NIE wykona

}

let counter = 0;

if (counter) {

// ten kod się NIE wykona

} else {

// ten kod się wykona

// ponieważ !! 0 === false

}

Tryb ścisły strict mode

Aby włączyć ten tryb, wystarczy na początku skryptu lub funkcji zastosować oznaczenie:

"use strict"; // dla całego skryptu

function fn () {

"use strict"; // tylko dla funkcji fn ()

}

Konieczność deklarowania zmiennych

Przy aktywnym trybie ścisłym nie można używać zmiennych, które nie zostały wcześniej zadeklarowane.

Duplikowanie parametrów funkcji

function overrideParameters(a, b, a) {

"use strict";

console.log(a) ;

}

overrideParameters(1, 2, 3);

SyntaxError : Duplicate parameter name not allowed in this context

Funkcje i obiekty

Czym są funkcje?

Jakieś operacje można zawrzeć w funkcji aby je póżniej móc wielokrotnie wykorzystywać w różnych miejscach kodu. Jeśli zajdzie potrzeba modyfikacji tych operacji to zmianę wykonamy tylko w jednym miejscu – tam gdzie zdefiniowaliśmy naszą funkcję.

Co więcej, funkcje można używać również wewnątrz innych funkcji. W danej funkcji wywołujemy jakąś funkcję, która zwróci nam wynik, który możemy zwrócić i przy okazji wykonać jakieś dodatkowe czynności, których nie chcieliśmy z jakiegoś powodu umieszczać w funkcji wewnętrznej a które w jakiś sposób ją dotyczą.

Zamknięcie operacji w prostą funkcję może dać realne korzyści, szczególnie w przypadku rozbudowywania i modyfikowania aplikacji. Istnieje nawet paradygmat w programowaniu, określany jako programowanie funkcyjne, gdzie jednym z założeń jest zamykanie praktycznie wszystkich operacji w małe funkcje.

Definiowanie i wywoływanie funkcji

Jest kilka sposobów definiowania funkcji. Warto jednak pamiętać, że nie ma jednej, konkretnej metody, która byłaby uznawana w świecie JavaScript za tę najlepszą. Wiele zależy tutaj od charakteru danej funkcji, sposobu jej używania oraz od praktyk przyjętych w zespole i w danej aplikacji.

Tworzenie funkcji

Deklarowanie funkcji poprzez słowo function w następujący sposób:

function name () {

// polecenia wewnątrz funkcji name

}

Funkcję wywołujemy:

name();

Innym sposobem stworzenia funkcji jest napisanie tzw. wyrażenia funkcyjnego w postaci:

const fun = function nameFunction () {

// polecenia wewnątrz funkcji name

};

Funkcję wywołujemy:

fun();

nameFunction(); Uncaught ReferenceError : nameFunction is not defined

Wywołanie funkcji po nazwie nameFunction powoduje zgłoszenie błędu, gdyż nasza funkcja jest dostępna pod nazwą zmiennej, do jakiej została przypisana, czyli w naszym przypadku fun

Przy tworzenia funkcji można pominąć nazwę funkcji, częściej się to robi gdy stosuje się wyrażenie funkcyjne. Używa się wtedy tzw. funkcji anonimowych:

const fn = function (a, b) {

return a + b;

};

lub zapisu określanego jako arrow function (tzw. funkcje strzałkowe), które zapisuje się jako:

const sum = (a, b) => {

return a + b;

};

Opisane tu trzy sposoby deklarowania funkcji nie są równoważne i czasami sposób i miejsce wywołania funkcji może decydować o konkretnym sposobie deklaracji.

Wartość zwracana przez funkcję

W języku JavaScript każda funkcja zawsze zwraca jakąś wartość albo w sposób jawny za pomocą słowa return albo niejawnie udefined.

Funkcja sprawdzające jakieś warunki i zwracające wartość typu boolean:

function isUserLogged (user) {

return isTokenValid(user.token) && user.role === 'ADMIN';

}

Funkcja może posiadać kilka instrukcji return:

function divide (a, b) {

if (b !== 0) {

return a / b; // instrukcja return kończy działanie funkcji

}

return "Nie dziel przez zero!";

}

Gdy instrukcja return zwraca jakiś obiekt.

function getUserPersonalData (user) {

return {

name: user.name,

age: user.age

};

}

// definiujemy przykładowego użytkownika :

const someUser = {

name: 'Tomek',

age: '35',

token: 'xyz'

};

getUserPersonalData(someUser); // { name : " Tomek ", age : " 35 "}

Czasami jednak tworzymy funkcje, których zadaniem jest wykonanie pewnych operacji:

function showUserAge(user.age) {

console.log('Wiek użytkownika: ' + user.age + ' lat.');

}

showUserAge(someUser.age); // Wiek użytkownika : 35 lat .

i wtedy nie jest ważne co ta funkcja zwraca.

Zakresy i domknięcia w JavaScript

Zakres zmiennej to zakres kodu programu, w którym dana zmienna jest dostępna.

Zaleca się deklarowanie zmiennych za pomocą słów let i const. Dzisiaj użycie słowa var może być wymagane tylko w aplikacjach, które muszą być uruchamiane w satrszych środowiskach

Zmienna, która została zadeklarowana przez let lub const , jest dostępna w kodzie dopiero od miejsca jej deklaracji. Próba użycia niezadeklarowanej zmiennej spowoduje rzucenie błędu.

function getPriceWithDiscount(basicPrice) {

  const totalPrice = basicPrice \* (1 - discount);

  const discount = 0.2;

*return* totalPrice;

}

console.log(getPriceWithDiscount(120));

*//ReferenceError: Cannot access 'discount' before initialization*

function getPriceWithDiscount(basicPrice) {

  var totalPrice = basicPrice \* (1 - discount);

  var discount = 0.2;

*return* totalPrice;

}

console.log(getPriceWithDiscount(120));

*// NaN; nie ma błędu jest zwrócona wartość NaN, która jest błędna*

W przypadku deklaracji z użyciem słowa var mamy do czynienia z tzw. zjawiskiem hoistingu i tak naprawdę powyższy kod wygląda pod spodem tak:

function getPriceWithDiscount(basicPrice) {

  var totalPrice; *// domyślnie undefined*

  var discount; *// domyślnie undefined*

  totalPrice = basicPrice \* (1 - discount);

*// tutaj mamy: number \* (1 – undefined) === NaN*

  discount = 0.2;

*// dopiero tutaj określamy wartość , ale nie ma ona już znaczenia*

*return* totalPrice; *// zwracamy wcześniej obliczone NaN*

}

console.log(getPriceWithDiscount(120)); *// NaN*

Hoisting polega na tym, że wszystkie deklaracje zmiennych z danego zakresu przechodzą na jego początek ze startową wartością undefined.

Bezpieczniej jest używać tylko deklaracje let lub const , które uchronią nas przed takimi sytuacjami, bo od razu otrzymamy w konsoli błąd a nie generowane niepoprawne wartości typu NaN, które nie pomagają zbytnio w zlokalizowaniu błędu.

W JS mamy trzy podstawowe rodzaje zakresu:

* Globalny
* Funkcyjny
* Blokowy

Istnieje jeszcze tzw. zakres modułu, często używany np. w pracy w środowisku Node.js

Zakres globalny

Zakres globalny oznacza, że zmienna jest dostępna w każdym miejscu kodu. Środowisko dodaje do zakresu globalnego obiekty np. window czy document. Jeśli zadeklarujemy zmienną za pomocą słowa kluczowego var poza jakąkolwiek funkcją i zakresem blokowym to znajdzie się w zakresie globalnym.

var variable = 40;

window.variable; // 40;

Zmienne deklarowane za pomocą słowa kluczowego let i const nie będą w zakresie globalnym.

const userAge = 40;

userAge; // 40

window.userAge; // undefined

Zmienne globalne są użyteczne gdy używamy bibliotek zewnętrznych(np. jQuery) lecz deklarowanie własnych zmiennych globalnych jest ryzykowne, gdyż wiąże się to z możliwością nadpisania zmiennej właśnie z tych bibliotek czy pola obiektu window.

Zakres funkcyjny

Oznacza on, że zmienna zadeklarowana wewnątrz funkcji jest dostępna tylko w zakresie tej funkcji.

const user = {

  name: 'Tomek',

  age: 40

};

function getName() {

  const prefix = 'Imię użytkownika';

*return* `${prefix} ${user.name}`;

}

console.log(getName());

console.log(prefix); *// ReferenceError: prefix is not defined*

Dobrą praktyką jest deklarowanie zmiennych i stałych tam, gdzie są one faktycznie potrzebne.

Argumenty funkcji są zmiennymi znajdującymi się w jej zakresie .

function addTwoNumbers(x) {

return function (y) {

return x + y;

}

}

addTwoNumbers(5)(2); // 7

Domknięcie, czyli zatrzaśnięcie dostępów do zmiennych i stałych w zakresie funkcyjnym z ich propagacją w dół. Funkcja anonimowa ma dostęp do zmiennej x zlokalizowanej w zakresie funkcji nadrzędnej. Do zmiennej y istnieje dostęp tylko z funkcji anonimowej propagacja dostępów nie działa w górę.

Aby obliczyć sumę dwóch liczb, musimy użyć dwukrotnego wywołania funkcji z dwukrotnym użyciem nawiasów okrągłych.

const sum = addTwoNumbers(5);

typeof sum; // "function"

sum(2); // 7

Powyższy przykład to tzw. domknięcie, gdzie zmienna x jest dostępna nawet po wywołaniu funkcji addTwoNumbers.

sum(6) // 11

Jeśli jakaś zmienna jest potrzebna tylko w jednej funkcji, to zadeklarujmy ją właśnie tam. Unikniemy przypadkowych błędów gdyby jakiś inny fragment kodu modyfikował zmienną globalną, z której my również korzystamy.

const sum = addTwoNumber(5);

console.log(sum(6)); *// 11*

console.log(sum(7)); *// 12*

console.log(addTwoNumber(9)(8)); *//17*

Zakres blokowy

Generalnie blokiem kodu nazywamy fragment zamknięty w nawiasy klamrowe.

{

const discount = 0.2;

const getPrice = price => price \* (1 – discount) ;

getPrice(100) // 80

}

getPrice(200); Uncaught ReferenceError : getPrice is not defined

Zakres blokowy dotyczy wyłącznie deklaracji z użyciem słowa const lub let.

{

const discount = 0.2;

function getPrice(price) {

return price \* (1 - discount);

}

getPrice(100); // 80

}

getPrice(200); // 160

Deklaracja funkcji podlega tzw. hoistingowi i jest przenoszona do zakresu globalnego, ale posiada ona dostęp do wartości stałej discount poprzez domknięcie (dotyczy zakresu blokowego, a nie funkcyjnego). Funkcję getPrice możemy wywołać poza blokiem.

Częściej będziemy używali zakresu blokowego w instrukcjach warunkowych czy pętlach.

function getPrice(price) {

let totalPrice;

if (price > 0) {

const discount = price > 100 ? 0.2 : 0.1;

totalPrice = price \* (1 - discount);

}

return totalPrice;

}

getPrice(100); // 90

getPrice(1000); // 800

let counter = 0;

for (let i = 0; i < 5; i++) {

counter += i;

}

counter; // 10

Funkcje, które od razu się wykonują

W JS można stworzyć funkcję wraz z jej natychmiastowym wykonaniem zwane IIFE, czyli Immediately-Invoked Function Expression.

Składnia konstrukcji IIFE:

(function () {

// instrukcje do wykonania

})();

(() => {

// instrukcje do wykonania

})();

Ich najczęstsze wykorzystanie, czyli tzw. wzorzec modułu:

const CustomModule = (function () {

const privateMethod = () => console.log('funkcja prywatna');

return {

publicFirstMethod: function () {

privateMethod();

console.log('pierwsza metoda publiczna');

},

publicSecondMethod: function () {

console.log('druga metoda publiczna');

}

};

})();

CustomModule.publicFirstMethod();

// funkcja prywatna

// pierwsza metoda publiczna

CustomModule.publicSecondMethod();

// druga metoda publiczna

CustomModule.privateMethod();

Uncaught TypeError : CustomModule . privateMethod is not a function

W momencie tworzenia obiektu CustomModule nasza funkcja IIFE od razu się wykonuje, więc od razu w CustomModule zostaje przypisany obiekt z dwiema metodami publicznymi. Cały czas mają one jednak dostęp do privateMethod, mimo że funkcja IIFE nie jest już nigdzie wywoływana ( patrz zakres widoczności i domknięcia).

Parametry domyślne funkcji

function getPriceWithDiscount(price, rebate) {

const discount = 1 - (rebate || 0.1); // domyślnie zakładamy 10% zniżki

return price \* discount;

}

getPriceWithDiscount(100, 0.2); // 80

getPriceWithDiscount(100); // 90

getPriceWithDiscount(100, 0); // 90 źle, naliczyliśmy rabat!

function getPriceWithDiscount(price, rebate) {

const discount = 1 - (typeof rebate === 'undefined' ? 0.1 : rebate);

return price \* discount;

}

getPriceWithDiscount(100, 0); // 100 , teraz wynik jest poprawny

getPriceWithDiscount(100, 0.3); // 70 , tutaj również obliczenie jest poprawne

Rozwiązanie ze sprawdzeniem typu wartości można spotkać w wielu starszych projektach i bibliotekach, dlatego warto je znać i wiedzieć jakie zagrożenia niesie ze sobą stosowanie alternatywy OR zamiast operatora typeof.

Określanie wartości domyślnych parametrów funkcji w nowych wersjach JavaScript:

function sum(a, b = 1, c) {

return a + b + c;

}

sum(2, 2, 2); // 6

sum(2, 0, 2); // 4

sum(2, 2); // 2 + 2 + undefined = 2 + 2 + Number(undefined) = 2 + 2 + NaN = NaN

sum(2, null, 2); // 2 + Number(null) + 2 = 2 + 0 + 2 = 4

sum(2, undefined, 2); // 2 + 1 + 2 = 5

Należy uważać przy wywoływaniu takich funkcji i w miejsce parametrów, które chcemy pominąć, wstawiać wartość undefined . Gdyby parametr z wartością domyślną był ostatnim parametrem wtedy chcąc skorzystać z wartości domyślnej, można po prostu pominąć parametr i wywołać funkcję z wartościami a i b.

Definiowanie obiektów

Zagnieżdzanie obiektów

const Person = {

name: 'Tomek',

age: 35,

address: {

city: 'Poznań',

country: 'Polska'

}

}

Trzy sposoby deklarowania metod:

const MathOperations = {

sum: function (a, b) {

return a + b;

},

subtraction: (a, b) => a - b,

divide(a, b) {

if (b === 0) {

return 'Nie dziel przez zero!';

}

return a / b;

}

}

MathOperations.sum(1, 2); // 3

MathOperations.subtraction(10, 4); // 6

MathOperations.divide(10, 2); // 5

MathOperations.divide(10, 0); // " Nie dziel przez zero

Modyfikowanie obiektów

Stwórzmy sobie testowy obiekt, który będziemy stopniowo modyfikować:

const Person = {

name: 'Tomek',

age: 35,

address: {

city: 'Poznań',

country: 'Polska'

}

}

Modyfikacja pola:

Person.name = 'Adam'; // "Adam"

Person['name'] = 'Adam'; // "Adam"

Modyfikacja zagnieżdżonego pola:

Person.address.city = 'Warszawa'; // "Warszawa"

Person['address']['city'] = 'Warszawa'; // "Warszawa"

const agePerson = 'age';

console.log(Person[agePerson]); // 35

Zabezpieczenie na wypadek próby odczytu nieistniejących pól.

if (Person.address && Person.address.country) {

Person.address.country = 'Szwecja';

}

Dodanie nowego pola do istniejącego obiektu:

Person.id = 123;

console.log(Person.id); // 123

Dodanie nowego obiektu, który będzie przechowywał kolejny obiekt, analogicznie jak nasze pole address:

Person.money = { bank: 1000, cash: 500 };

Person.age = undefined; // nie usunie pola tylko nada mu wartość undefined

'age' in Person; // true

Pole możemy usunąć operatorem:

delete Person.age;

'age' in Person; // false

Słowo const tworzy w odniesieniu do obiektów zmienną, której nie można nadpisać, ale można ją modyfikować. Dlatego modyfikacje obiektu Person są dozwolone, ale próba nadpisania Person nowym obiektem będzie błędem:

Person = { name: 'Adam', age: 40 }

Uncaught TypeError : Assignment to constant variable

Dla typów prostych deklaracja ze słowem const oznacza stałą:

const name = 'Tomek';

name = 'Adam';

Uncaught TypeError : Assignment to constant variable .

name[0] = 'A'; // zmiana pierwszego znaku stałego napisu nie wywoła błędu ale będzie nie skuteczna

name; // "Tomek"

Operatory spread i rest

Operatory spread i rest (ECMAScript 6) upraszczają zapis. Oba operatory mają taki sam symbol (...), a ich znaczenie zależy od miejsca użycia.

Operator spread służy do rozbicia obiektu lub tablicy na poszczególne elementy.

Przykłady użycia operatora spread

const A = {

  x: 5,

  y: 10,

};

const B = {

  z: 50,

};

const point = {

  ...A,

  ...B,

};

console.log(point); *// { x: 5, y: 10, z: 50 }*

point.x = 13;       *// to tylko kopia pola x*

console.log(A.x); *// 5*

const C = {

  coordinate: {

    x: 6,

    y: 7,

  },

  id: 123,

};

const D = {

  ...C.coordinate,

}

console.log(D); *// { x: 6, y: 7 }*

Operator spread jest także używany w odniesieniu do tablic.

Programowanie zorientowane obiektowo w języku JavaScript. Ved Antani, Stoyan Stefanov

Proste typy danych, tablice, pętle i warunki

Zmienne

Zmienne służą do przechowywania danych. Zmienne stosujemy także wtedy, gdy warość danych nie jest znana podczas pisania kodu, np. gdy wartość jest wynikiem działania przeprowadzonego już po uruchomieniu programu.

Ze zmiennej możemy skorzystać gdy:

* Zadeklarujemy zmienną
* Zainicjalizujemy ją, czyli nadamy jej poczatkową wartość

Do deklaracji zmiennych służy słówko var.

*// Przykłady deklaracji zmiennych za pomocą słowa kluczowego var*

var a;

var thisIsVaraiable;

var \_and\_this\_too;

var mix12three;

Nazwy zmiennych mogą się skladać z dowolnej kombinacji liter, cyfr, podkreślnika oraz znaku dolara. Nie mogą się jednak zaczynać od cyfry.

Inicjowanie zmiennej to nadanie jej wartości po raz peirwszy. Można to zrobić:

* Najpierw zadeklarować zmienną a potem zainicjalizować;
* Zadeklarować i zainicjalizować zmienną za pomocą pojedyńczej instrukcji.

*// Deklarowanie kilku zmiennych za pomocą jednej instrukcji*

var v1, v2, v3 = 'witaj', v4 = 4, v5;

*// Dla zwiększenia czytelności zapisujemy tak*

var var1,

  var2,

  var3 = 'żegnaj',

  var4 = 4,

  var5;

**Znak $ w nazwach zmiennych**

Ten znak może się pojawić w dowolnym miejscu zmiennej. Poprzednie wersje standartu ECMA zalecały stosowanie go tylko w kodzie wygenerowanym, jednakże społecznośc JS to ignoruje.

Wielkość liter w nazwach zmiennych ma znaczenie

W nazwach zmiennych rożrózniane sa wielkie i małe litery.

var case\_matters = 'małe';

var CASE\_MATTERS = 'duże';

console.log(case\_matters); *// małe*

console.log(CASE\_MATTERS); *// duże*

Kod wpisywany w konsoli czasem powoduje wyświetlenie warości undefine. Dzieje się tak podczas ewaluacji (wykonywania) wpisywanego kodu – konsola drukuje zwracaną warość. Takie wyrażenia jak var a = 1 nie zwracają beżpośrednio nic, dlatego jest drukowane undefined, ale case\_matters tak.

Operatory

Operatory pobierają jako argumenty jedną warość ablo dwie (lub zmienne), wykonują na nich pewną operację i zwaracają watość

var a1 = 2; *// instrukcja przypisania, Znak = to operator przypisania*

var a2 = 3;

var a3;

*// Prosty przykład użycia operatora*

a3 = a1 + a2; *// Znak + jest operatorem;*

Uzgodnienie terminologi:

* Znak + jest operatorem
* Operacją (działaniem) jest dodawanie
* Wartości wejściowe (inaczej argumenty ) - są przechowywane w zmiennych a1, a2 to 2 i 3
* Wynikiem jest liczba 5 przechowywana w zmiennej a3
* Całość jest zwana wyrażeniem

Podstawowe symbole operatorów artmetycznych to: +, -, \*, /, % a operacje wykonywane za pomocątych operatorów to odpowiednio: dodawanie, odejmoanie, mnożenie, dzielenie i modulo czyli reszta z dzielenia.

*/ Sprawdzenie czy liczba jest parzysta*

var even;

var number = prompt('Podaj liczbę całkowitą');

parseInt(number) % 2 ? even = 'nieparzysta' : even = 'parzysta';

alert(`Podałeś liczbę: ${number}, która jest ${even}`);

Proste typy zmiennych

Proste typy zmiennych – podsumowanie

Tablice

Warunki i pętle

Bloki kodu

Pętle

Komentarze

Ćwiczenia

Podsumowanie

Funkcje

Tam gdzie w innych językach trzeba zastosować obiektowość JS udostępnia funkcje, dzięki którym jest tak elastyczny i ekspresywny.

Czym są funkcje?

Funkcja jest to zgrupowana pewna ilość kodu posiadająca nazwę i która może być ponownie użyta za pomocą tejże nazwy.

Przykład:

function sum(a, b) {

let c = a + b;

return c;

}

Funkcja składa się z:

* Słowa kluczowego function
* Nazwy funkcji – tu sum
* Parametrów – tu a, b. Funkcja może mieć dowolną ilość parametrów rozdzielonych przecinkami albo wcale
* Bloku kodu zwanego ciałem funkcji
* Instrukcji return. Funkcja zawsze zwraca wartość. Jeśli nie ma instrukcji return to niejawnie zwraca undefined

Można zwrócić więcej niż jedną wartość zwracając tablicę warości.

Ogólna postać deklaracji funkcji:

function nameFunction(param1, param2, ... , paramN){

statement1;

statement2;

…;

statementN;

return value;

}

Wywołanie funkcji:

nameFunction(arg1, arg2, … , argN);

Jeśli wywołamy funkcję a ona zwraca jakąś wartość to możemy ją przypisać do jakieś zmiennej aby ją póżniej móc wykorzystać:

let result = sum(1, 3);

console.log(result); // 3

Parametry

Funkcja nie musi przyjmować parametrów, ale jeśli ich oczekuje i zapomnisz podać je podczas wywoływania, JavaScript przypisze im wartość undefined .

sum(1) // NaN = 1 + undefined

Mimo iż istnieje technicznie rzecz biorąc różnica między parametrami a argumentami, te dwa terminy są używane zamiennie, jak zresztą wiele innych pojęć w informatyce np. definicja i deklaracja. Parametry są definiowane razem z funkcją, podczas gdy argumenty są przekazywane do funkcji w trakcie jej wywołania.

Jeśli przekażemy więcej parametrów, niż funkcja oczekuje, to ich nadmiar zostanie zignorowany:

sum(1, 2, 3, 4, 5); //3

Możliwe jest pisanie funkcji, które mogą przyjmować różną liczbę parametrów a to dzięki temu, że w każdej funkcji jest tworzona specjalna wartość arguments. Oto funkcja, której działanie polega na zwracaniu wszystkich przekazanych jej argumentów:

function args() {

return arguments;

}

args(); // []

args(1, 2, 3, true, ‘marco’); // [1, 2, 3, 4, true, "marco"]

Użyj arguments aby poprawić funkcję sum () tak, by przyjmowała dowolną liczbę parametrów i dodawała je wszystkie.

function sumOnSteroids() {

let res = 0;

for (let i = 0; i < arguments.length; i++) {

res += arguments[i];

}

return res;

}

sumOnSteroids(1, 2, 3) // 6

sumOnSteroids(1, 2, 3, 4, 4, 3, 2) // 19

sumOnSteroids() // 0

Parametry domyślne

Parametrom funkcji można przypisać wartości domyślne. W przypadku pominięcia parametru podczas wywoływania funkcji używana jest domyślna wartość przypisana do tego parametru:

function render(fog\_level=20, spark\_level=100){

console.log(`Poziom zamglenia: ${fog\_level}, a natężenie iskier: ${spark\_level}`)

}

render(10); // Poziom zamglenia: 10, a natężenie iskier: 100

Warto zauważyć, że undefined uważa się za brak wartości parametru.

render(undefined,10); // Poziom zamglenia: 20, a natężenie iskier: 10

Podczas określania domyślnych wartości parametrów można również odwoływać się do innych parametrów:

function render(fog\_level=1, spark\_level=fog\_level \* 10){

console.log(`Poziom zamglenia: ${fog\_level}, a natężenie iskier: ${spark\_level}`)

}

render(10) // Poziom zamglenia: 10, a natężenie iskier: 100

Parametry domyślne mają własny zakres. Zakres ten jest umieszczony pomiędzy zewnętrznym i wewnętrznym zakresem funkcji. Jeśli parametr jest przysłaniany przez zmienną w zakresie wewnętrznym, zaskakująco zmienna wewnętrzna nie jest dostępna.

var scope="zakres zewnętrzny";

function scoper(val=scope){

var scope="zakres wewnętrzny";

console.log(val);

}

scoper(); // zakres zewnętrzny

Można spodziewać się, że val zostanie przesłonięte przez wewnętrzną definicję zmiennej scope , ponieważ jednak domyślne parametry mają swój własny zakres, wewnętrzny zakres nie ma wpływu na wartość przypisaną do val .

function sum(a = 5, b = 6) {

  return a + b;

}

console.log(sum(49)); // 55

console.log(sum(undefined, 49)); 54

function sumOne(a = 5, b){

  return a + b;

}

console.log(sumOne(51)); // NaN = 51 + undefined

console.log(sumOne(undefined, 51)); //56

function sumTwo(a = 6, b = a){

  return a + b;

}

console.log(sumTwo(44)); // 88

console.log(sumTwo(undefined)); // 12

Parametry reszty

Parametry reszty pozwalają wysyłać do funkcji dowolną liczbę parametrów w postaci tablicy. Parametrem reszty może być tylko ostatni na liście parametrów i może być tylko jeden parametr reszty. Umieszczenie operatora reszty ( ... ) przed ostatnim formalnym parametrem wskazuje, że ten parametr jest parametrem reszty.

function sayThings(tone, ...quotes){

console.log(Array.isArray(quotes)); // true

console.log(`Głosem ${tone} mówię: ${quotes}`)

}

sayThings("Morgana Freemana","Coś poważnego", " Implodujący Wszechświat"," Amen");

// Głosem Morgana Freemana mówię: Coś poważnego, Implodujący Wszechświat, Amen

Parametry reszty są tablicą i dostępne są dla nich wszystkie metody tablicowe. Zmienne argumenty ( var-args ) są od dawna częścią kilku innych języków i są mile widzianą zmianą w ES6.

function sumVar(...array){

  let result = 0;

  for(let i = 0; i < array.length; i++){

    result += array[i];

  }

  return result;

}

console.log(sumVar(1, 3, 5, 7, 9)); // 25

Operator rozwijania

Operator rozwijania (ang. spread operator ) wygląda dokładnie tak samo jak operator reszty. Operatory rozwijania są używane, gdy dostarczamy argumentów podczas wywoływania funkcji lub definiowania tablicy. Operator rozwijania przyjmuje tablicę i dzieli jej elementy na poszczególne zmienne.

function sumAll(a,b,c){

return a + b + c

}

var numbers = [6,7,8]

// Sposób przekazywania tablicy jako argumentu funkcji w ES5

console.log(sumAll.apply(null,numbers)); // 21

// Operator rozwijania w ES6

console.log(sumAll(...numbers)); // 21

Operatory rozwijania zwiększają możliwości pracy z tablicami w JavaScripcie. Jeśli chcesz utworzyć tablicę zawierającą elementy innej tablicy, istniejąca składnia tablicy tego nie obsługuje. Aby to osiągnąć, musielibyśmy użyć metod push , splice i concat . Jednak dzięki operatorom rozwijania staje się to trywialne:

var midweek = ['Śr', 'Czw'];

var weekend = ['Sob', 'Niedz'];

var week = ['Pon','Wt', ...midweek, 'Pt', ...weekend];

console.log(week); // ["Pon","Wt","Śr","Czw","Pt","Sob","Niedz"]

const array = [2, 33, 3, 44, 4, 55];

console.log(sumVar(...array)); // 141 ; operator rozwijania

Funkcje predefiniowane

Istnieje pewna liczba funkcji, które zostały wbudowane w silnik JavaScriptu i z których można korzystać do woli:

* parseInt ()
* parseFloat ()
* isNaN ()
* isFinite ()
* encodeURI ()
* decodeURI ()
* encodeURIComponent ()
* decodeURIComponent ()

Zasada czarnej skrzynki

Z reguły podczas wywoływania funkcji Twój program nie musi wiedzieć, jakie czynności są wykonywane wewnątrz danej funkcji. Możesz potraktować funkcje jak czarne skrzynki — podajesz im pewne wartości (w postaci argumentów wejściowych) i odbierasz od nich zwracane wyniki. Jest to prawdziwe dla wszystkich funkcji — tych wbudowanych w silnik JavaScriptu, pisanych przez Ciebie czy utworzonych przez Twoich współpracowników lub nieznanych Ci programistów.

parseInt()

Funkcja parseInt () przyjmuje dane wejściowe dowolnego typu (najczęściej łańcuch znaków) i próbuje przekonwertować je na liczbę całkowitą. Jeśli operacja nie powiedzie się, zwracana jest wartość NaN.

parseInt('123'); //123

parseInt('abc123'); // NaN

parseInt('1abc23'); //1

parseInt('123abc'); //123

Funkcja przyjmuje jeszcze opcjonalny drugi argument, który określa podstawę, opisującą typ liczby: dziesiętny, szesnastkowy, binarny itd.

parseInt(‘FF’, 10); // NaN

parseInt('FF', 16); // 255

parseInt('0377', 10); // 377

parseInt('0377', 8); // 255

Jeśli drugi argument nie zostanie podany, za podstawę uznawana jest liczba 10 , z następującymi wyjątkami:

* Jeśli jako pierwszy argument przekazany zostanie łańcuch zaczynający się od 0x , drugiemu argumentowi (jeśli nie został podany) przypisana zostanie wartość 16 (liczba zostanie uznana za szesnastkową).
* Jeśli pierwszy parametr zaczyna się od 0 , drugi otrzyma wartość 8.

parseInt('377'); // 377

parseInt('0377'); // 255

parseInt('0x377'); // 887

Najbezpieczniejszym rozwiązaniem jest określanie podstawy za każdym razem. Wyobraź sobie na przykład, że parsujesz pola formularza, który reprezentuje kalendarz, a użytkownik wpisał 08 lub 06.

ECMAScript 5 usuwa prefiks 0 z zapisu liczb ósemkowych i pozwala uniknąć zamieszania z parseInt () i nieokreśloną podstawą.

parseInt('0377'); // 377

parseInt('0377', 8); // 255

parseFloat()

Funkcja parseFloat () działa podobnie do parseInt (), ale w danych wejściowych szuka ułamków dziesiętnych. Przyjmuje tylko jeden parametr.

parseFloat('123'); //123

parseFloat('1.23'); //1.23

parseFloat('1.23abc.00'); // 1.23

Podobnie jak parseInt () , parseFloat () poddaje się po napotkaniu pierwszego znaku, z którym nie umie sobie poradzić, nawet jeśli pozostała część tekstu zawiera poprawne liczby.

parseFloat('a.bc1.23'); // NaN

Funkcja parseFloat () , w przeciwieństwie do parseInt () , jest w stanie poprawnie zinterpretować zapis wykładniczy.

parseFloat('123e-2'); //1.23

parseFloat('1e10'); //10000000000

parseInt('1e10'); //1

isNaN()

Za pomocą isNaN () można sprawdzić, czy wartość wejściowa jest liczbą.

isNaN(NaN); true

isNaN(123); false

isNaN(1.23); false

isNaN(parseInt('abc123')); true

Ta funkcja stara się również przekonwertować dane wejściowe na liczbę:

isNaN('1.23'); false

isNaN('a1.23'); true

Funkcja isNaN () jest użyteczna także dlatego, że specjalna wartość NaN nie jest równa niczemu, nawet samej sobie. Wynikiem porównania NaN === NaN będzie false . Dlatego NaN nie może być używana do sprawdzania, czy dana wartość jest prawidłową liczbą.

isFinite()

Funkcja isFinite () sprawdza, czy dane wejściowe to liczba różna od Infinity i różna od NaN.

isFinite(Infinity); //false

isFinite(-Infinity); //false

isFinite(12); //true

isFinite(1e308); //true

isFinite(1e309); //false

Jeśli dziwią Cię dwa ostatnie wyniki, przypominamy, że największą dopuszczalną liczbą w języku JavaScript jest 1.7976931348623157e + 308 , więc 1e309 jest w efekcie nieskończonością.

encodeURI() i encodeURIComponent()

W adresach URL (ang. Uniform Resource Locator ) i URI (ang. Uniform Resource Identifier ) niektóre znaki mają specjalne znaczenie. Jeśli chcemy mieć pewność, że nie będą one interpretowane, możemy skorzystać z funkcji encodeURI () lub encodeURIComponent () . Pierwsza z nich zwraca poprawny adres URL, a druga zakłada, że przekazujemy tylko część adresu URL (np. parametry żądania), i koduje odpowiednie znaki.

var url = 'http://www.packtpub.com/scr ipt.php?q=this and that';

encodeURI(url);

//"http://www.packtpub.com/scr%20 ipt.php?q=this%20and%20that"

encodeURIComponent(url);

//"http%3A%2F%2Fwww.packtpub.com%2Fscr%20ipt.php%3Fq%3Dthis% 20and%20that"

let url = "https://raubuc.net"

console.log(encodeURI(url));

eval()

Funkcja eval () przyjmuje łańcuch znaków i wykonuje go jako kod JavaScriptu:

eval('var ii = 2;');

ii //2

Są sytuacje, w których eval () się przydaje, jednak w miarę możliwości należy tej funkcji unikać.

Weterani JavaScriptu jak mantrę powtarzają zdanie „ eval is evil” ( eval to samo zło).

Bonus — funkcja alert()

Nie należy ona do rdzenia języka (nie ma jej w specyfikacji ECMA), ale jest obsługiwana przez środowisko hosta, czyli przeglądarkę. Pozwala ona na wyświetlanie komunikatów w okienku dialogowym.

Okienko dialogowe blokuje wątek przeglądarki, co oznacza, że żaden inny kod nie zostanie wykonany, zanim użytkownik nie zamknie alertu. Jeśli aplikacja jest często aktualizowaną aplikacją AJAX, używanie funkcji alert () nie jest najlepszym pomysłem.

Zakres zmiennych

Zmienne w JS nie są definiowane w zakresie bloku, tylko w zakresie funkcji. Oznacza to, że jeśli zmienna została zdefiniowana wewnątrz funkcji, nie jest widoczna poza nią. Natomiast zmienna zdefiniowana wewnątrz bloku if lub for jest widoczna poza blokiem. Zmienne globalne to zmienne definiowane poza funkcjami (w globalnym kodzie programu), natomiast zmienne lokalne to zmienne definiowane wewnątrz funkcji. Kod wewnątrz funkcji ma dostęp zarówno do zmiennych globalnych, jak i do swoich zmiennych lokalnych.

var global = 1;

function f() {

var local = 2;

global += local;

return global;

}

f() // 3

global // 3

local ;

Uncaught ReferenceError: local is not defined

Najlepsze praktyki Staraj się ograniczać liczbę zmiennych globalnych, aby uniknąć konfliktów nazw. Wyobraź sobie, że dwie osoby pracują nad dwiema różnymi funkcjami w tym samym skrypcie i przypadkowo postanawiają nadać tę samą nazwę zmiennej globalnej. Zawsze deklaruj zmienne za pomocą instrukcji var . Rozważ stosowanie wzorca pojedynczej instrukcji var . Wszystkie zmienne potrzebne w funkcji definiuj na samej górze funkcji, żebyś mógł zawsze znaleźć zmienne w tym samym miejscu i zapobiegać powstawaniu przypadkowych zmiennych globalnych.

Ponadto należy mieć na uwadze, że jeśli do zadeklarowania zmiennej nie zostanie użyte słowo kluczowe var , tej zmiennej zostanie automatycznie przypisany zakres globalny.

function f() {

local = 2;

}

local;

Uncaught ReferenceError: local is not defined

f() ;

local; // 2

Funkcja f () zawiera zmienną local . Przed wywołaniem funkcji zmienna nie istnieje. Jednak podczas pierwszego wywołania funkcji zmienna jest tworzona i ma zasięg globalny. Dlatego jeśli wówczas spróbujemy sięgnąć do zmiennej local , okaże się ona dostępna.

Powyższe uwagi ,w związku z tym że zalecane jest nie używać do deklarowania słowa kluczowego var lecz let i const, poniekąd się zdeaktualizowały.

Wynoszenie zmiennych

var a = 123;

function f() {

alert(a);

var a = 1;

alert(a);

}

f();

Wewnątrz funkcji zakres lokalny jest ważniejszy od globalnego. Zmienna lokalna nadpisuje każdą zmienną globalną o tej samej nazwie.

Kiedy wykonywanie programu JavaScript wchodzi w nową funkcję, wszystkie zmienne zadeklarowane w różnych miejscach funkcji są przesuwane, czyli wynoszone w górę funkcji (jest to tzw. z ang. hoisting czyli wynoszenie zmiennych). Jednakże przypisania pozostają tam gdzie były. To jakby powyższa funkcja była napisana tak:

var a = 123;

function f() {

var a; // ⬄ var a = undefined ;

alert(a); // undefined

a = 1;

alert(a); // 1

}

Można również przyjąć wzorzec pojedynczej instrukcji var . W takim przypadku będziemy robić coś na kształt ręcznego wynoszenia zmiennych, aby zapobiec nieporozumieniom związanym z zachowaniem mechanizmu wynoszenia języka JavaScript.

Zakres bloku

Jeżeli kodujesz w ES6, zakres bloku w większości zastąpi Twoją potrzebę używania zmiennych zadeklarowanych za pomocą var . Jeśli jednak nadal piszesz w ES5, powinieneś uważnie przyjrzeć się zachowaniu mechanizmu wynoszenia.

ES6 wprowadza słowa kluczowe let i const , które pozwalają deklarować zmienne.

Zmienne zadeklarowane za pomocą let mają zakres bloku. Istnieją tylko w bieżącym bloku kodu.

var a = 1;

{

let a = 2;

console.log( a ); // 2

}

console.log( a ); // 1

Zaleca się, aby podczas deklarowania zmiennych o zakresie blokowym ogólnie dodawać deklarację let na początku bloku.

function swap(a, b){ // <-- tu zaczyna się zakres funkcji

if(a > 0 && b > 0){ // <-- tu zaczyna się zakres bloku

let tmp = a;

a = b; b = tmp;

} // <-- tu kończy się zakres bloku

console.log(a, b);

console.log(tmp);

// zmienna tmp nie jest zdefiniowana , ponieważ jest dostępna tylko w zakresie bloku

return [a,b];

}

swap(1,2);

Otóż przy użyciu słowa kluczowego let nie można ponownie zadeklarować tej samej zmiennej w tej samej funkcji lub zakresie bloku.

Kolejnym słowem kluczowym wprowadzonym w ES6 jest const (stała). Zmienna zadeklarowana za pomocą słowa kluczowego const tworzy dla zmiennej referencję tylko do odczytu. Nie oznacza to, że wartość przechowywana przez tę referencję jest niemutowalna. Nie można jednak ponownie przypisać identyfikatora zmiennej. Stałe mają zakres bloku tak samo jak zmienne utworzone za pomocą słowa kluczowego let . Ponadto podczas deklarowania trzeba przypisać wartość do zmiennej.

Chociaż brzmi to dziwnie, const nie ma nic wspólnego z wartościami niemutowalnymi. Stałe tworzą niemutowalne wiązanie. Jest to ważne rozróżnienie i należy je prawidłowo zrozumieć. Rozważmy następujący przykład:

const car = {}

car.tyres = 4

Jest to poprawny kod. Przypisujemy w nim wartość {} do stałej car . Po przypisaniu tej referencji nie można zmienić.

W ES6 należy stosować się do następujących zasad:

* Używaj słowa kluczowego const w miarę możliwości. Używaj go dla wszystkich zmiennych, których wartości się nie zmieniają.
* Używaj słowa kluczowego let .
* Unikaj słowa kluczowego var .

Zakres zmiennych – moje spostrzeżenia

Możliwe są takie deklaracje bez rzucania błędów:

var zmienna = 4;

console.log(zmienna); // 4

var zmienna = 'zmienna';

console.log(zmienna); // zmienna

var zmienna = false;

console.log(zmienna); // false

Natomiast taki zestaw deklaracji:

var zmienna = 4;

console.log(zmienna);

let zmienna = 'zmienna';

console.log(zmienna);

spowoduje:

Uncaught SyntaxError: Identifier 'zmienna' has already been declared

Taki zestaw deklaracji też nie będzie powodował błędów:

var zmienna = 4;

console.log(zmienna); // 4

var zmienna = 'zmienna';

console.log(zmienna); // zmienna

{

  let  zmienna = false;

  console.log(zmienna); // false

}

Taki tak:

{

  let  zmienna = false;

  console.log(zmienna);

  let zmienna = 'zmienna';

  console.log(zmienna);

}

Uncaught SyntaxError: Identifier 'zmienna' has already been declared

Stąd wniosek że za pomocą słowa kluczowego var można zduplikować deklarację zmiennej w tym samym zakresie ale już za pomocą słowa let nie.

Rozpatrzmy taki zestaw instrukcji:

function myFunction () {

  zmienna = 'zmienna';

  console.log(zmienna); // zmienna

  var zmienna = false;

  console.log(zmienna); // false

}

myFunction();

console.log(zmienna); Uncaught ReferenceError: zmienna is not defined

Dlaczego zmienna nie jest dostępna poza funkcją przecież została tam zadeklarowana bez słowa var czyli jest zmienną globalną. Jednakże druga deklaracja ze słowem var ją nadpisała, został zapewne zastosowany hoisting i deklaracja var zmienna została wyniesiona na początek zakresu funkcji.

Natamiast dla takich instrukcji :

function myFunction () {

  zmienna = 'zmienna';

  console.log(zmienna); // zmienna

  zmienna = false;

  console.log(zmienna); // false

}

myFunction();

console.log(zmienna); // false

Zmienna zmienna jest zmienną globalną.

Funkcje są danymi.

W JS funkcje są danymi a więc można je przypisać do zmiennej:

var f = function () { return 1; };

Ten sposób definiowania funkcji - **zapisem literałowym funkcji**.

Fragment function () { return 1 ;} jest **wyrażeniem funkcyjnym** (ang. function expression ). Może ono mieć nazwę - **nazwanym wyrażeniem funkcyjnym** (ang. named function expression — NFE).

typeof f // ”function”

Funkcje są danymi, które :

* zawiera kod,
* jest wykonywalny (funkcje mogą być wywoływane).

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

var add = sum;

typeof add; // “function”

add(1, 2); // 3

Funkcje anonimowe

Funkcja anonimowa to funkcja, która nie ma nazwy:

function() {

// instrukcje do wykonania

}

Zastosowania dla funkcji anonimowych:

* Funkcję anonimową można przekazać jako parametr do innej funkcji. Funkcja odbierająca może z przekazaną funkcją zrobić coś pożytecznego.
* Funkcje anonimowe można definiować i od razu wykonywać.

Wywołania zwrotne

Jeśli funkcja A zostaje przekazana funkcji B, a następnie B wywołuje A, często mówi się, że A jest funkcją **wywołania zwrotnego** (ang. callback function ). Jeśli A nie ma nazwy, to możemy powiedzieć, że jest anonimowym wywołaniem zwrotnym.

function B(A) {

A();  
}

function call(callback){

callback();

}

Zalety wywołań zwrotnych:

* Wywołania zwrotne umożliwiają przekazywanie funkcji bez konieczności ich nazywania, co oznacza, że potrzebnych jest mniej zmiennych.
* Możemy przenieść obowiązek wywołania funkcji na inną funkcję, co oznacza, że musimy napisać krótszy kod.
* Wywołania zwrotne mogą korzystnie wpłynąć na wydajność aplikacji poprzez opóźnianie wykonyywania lub odblokowywanie wywołań.

Przykłady wywołań zwrotych

function invokeAdd(a, b) {

  return a() + b();

}

function one(){

  return 1;

}

function two(){

  return 2;

}

console.log(invokeAdd(one, two));

console.log(invokeAdd(()=>{return 2}, ()=>{return 2}));

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

function multiplyByTwo(...array){

  const ar = [];

  for(let i =  0; i < array.length; i++){

    ar[i] = array[i] \* 2;

  }

  return ar;

}

function addOne(a){

  return a + 1;

}

console.log(multiplyByTwo(4, 6, 7, 8, 11));

console.log(addOne(100));

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

const myarr = multiplyByTwo(10, 20, 30);

console.log(myarr);

for (let i = 0; i < myarr.length; i++) {

  myarr[i] = addOne(myarr[i]);

}

console.log(myarr);

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

function multiplyByTwoVersionOne(callback, ...array){

  const arr = [];

  for (let i = 0; i < array.length; i++) {

    array[i] = callback(array[i] \* 2);

  }

  return array;

}

console.log(multiplyByTwoVersionOne(addOne, 2, 6, 8, 9, 33));

console.log(multiplyByTwoVersionOne((a)=>{return a + 2}, 2, 6, 8, 9, 33));

Moje wywołanie zwrotne:

let counter = 0

setInterval(() => {

  const container = document.getElementsByClassName('container');

  counter++;

  switch (counter % 5) {

    case 0: container[0].style.flexDirection = 'row';

      break;

    case 1: container[0].style.flexDirection = 'row-reverse';

      break;

    case 2: container[0].style.justifyContent = 'flex-end';

      break;

    case 3: container[0].style.flexDirection = 'column';

      break;

    case 4: container[0].style.flexDirection = 'column-reverse';

      break;

  }

  if (counter == 1000) {

    counter = 0;

  }

}, 3000);

Funkcje natychmiastowe

Inne zastosowanie funkcji anonimowej to wywoływanie funkcji zaraz po jej zdefiniowaniu. Wyrażenie funkcyjne umieszczamy w nawiasach i dodajemy kolejną parę nawiasów oznaczająca natychmiastowe wykonanie a w nich ewentualne parametry :

(function(name){

alert('Witaj, ' + name + '!');

})('stary');

Alternatywnie można przenieść zamknięcie pierwszej pary nawiasów na koniec.

(function () {

// ...

}());

Kod zostanie wykonany bez tworzenia nadmiaru zmiennych globalnych. Tej samej funkcji nie da się wykonać dwukrotnie. Dlatego anonimowe funkcje samowywołujące najlepiej nadają się do wykonywania zadań jednorazowych lub inicjujących.

Funkcja natychmiastowa może również zwracać wartość.

var result = (function () {

// robimy coś skomplikowanego

// z tymczasowymi zmiennymi lokalnymi ...

// ...

// coś zwracamy ;

}());

Funkcje wewnętrzne (prywatne)

Możemy zdefiniować funkcję wewnątrz innej funkcji.

function outer(param) {

function inner(theinput){

return theinput \* 2;

}

return 'Wynik wynosi ' + inner(param);

}

Za pomocą wyrażenia funkcyjnego możemy również zapisać to tak:

var outer = function (param) {

var inner = function (theinput) {

return theinput \* 2;

};

return 'Wynik wynosi ' + inner(param);

};

Kiedy wywołana zostanie globalna funkcja outer () , wewnętrznie wywoła również lokalną funkcję inner () . Ponieważ funkcja inner () jest lokalna, nie jest dostępna poza outer () i dlatego nazywamy ją funkcją prywatną.

outer(2); //Wynik wynosi 4

outer(8); //Wynik wynosi 16

inner(2); ReferenceError: inner is not defined

Ze stosowania funkcji prywatnych płyną następujące korzyści:

* Nie dochodzi do zaśmiecenia globalnej przestrzeni nazw, co zmniejsza ryzyko powstawania kolizji nazw.
* Prywatność — na zewnątrz widoczne są tylko te funkcje, które programista chce udostępnić. Funkcjonalności nieprzeznaczone dla reszty aplikacji są ukryte.

Funkcje, które zwracają funkcje

Funkcja zawsze zwraca wartość albo za pomocą instrukcji return albo domyślnie wartość undefined . Ponieważ funkcja jest daną to można ją zwrócić przez inna funkcję.

function date() {

  let time = new Date();

  return function(){

   let hours = time.getHours();

   let minutes = time.getMinutes();

   let secondes = time.getSeconds();

    if (secondes < 10) {

      secondes = '0' + secondes;

    }

    if (minutes < 10) {

      minutes = '0' + minutes;

    }

    if (hours < 10) {

      hours = '0' + hours;

    }

    const clock = document.getElementById('clock');

    clock.innerHTML = hours + ':' + minutes + ':' + secondes;

  }

}

setInterval(() => {

 /\* const time = date();

  time(); ⬄ \*/

date()();

}, 1000);

Funkcjo, przepiszże się!

Ponieważ funkcje potrafią zwracać funkcje, możliwe jest zastąpienie oryginalnej funkcji tą zwracaną.

function a() {

alert('A!');

return function(){

alert('B!');

};

}

Wartość zwróconą przez wywołanie a () można przypisać zmiennej a, nadpisując w ten sposób istniejącą funkcję:

a = a();

Powyższa linia kodu przy pierwszym wykonaniu spowoduje wyświetlenie ' A !' , jednak następne wywołanie a() wyświetli ' B !' . Opisany mechanizm jest przydatny, jeśli funkcja wykonuje pewne jednorazowe zadanie. Po pierwszym wywołaniu funkcja nadpisuje się, aby uniknąć niepotrzebnego wykonywania określonej pracy przy każdym jej wywołaniu.

W powyższym przykładzie funkcja została przedefiniowana z zewnątrz, a zwracana wartość została przypisana do funkcji. Funkcja może jednak przepisać się sama z wewnątrz, tak jak pokazano poniżej:

function a() { alert('A!'); a = function(){ alert('B!'); }; }

Przy pierwszym wywołaniu ta funkcja wykona następujące czynności:

* Wyświetli ' A !' (uznajmy to za nasze jednorazowe zadanie inicjujące).
* Przedefiniuje globalną zmienną a , przypisując do niej nową funkcję. Każde kolejne wywołanie będzie powodowało wyświetlenie ' B !' .

Oto inny przykład, który łączy kilka technik omówionych:

var a = (function () {

function someSetup() {

var setup = 'zrobione';

}

function actualWork() {

alert('Praca wre');

}

someSetup();

return actualWork;

}());

W przykładzie należy zwrócić uwagę na następujące kwestie:

* Mamy funkcje prywatne: someSetup () i actualWork () .
* Mamy funkcję natychmiastową w postaci funkcji anonimowej, która wywołuje samą siebie za pomocą pary nawiasów umieszczonych po jej definicji.
* Pierwsze wykonanie polega na wywołaniu funkcji someSetup () i zwróceniu referencji do zmiennej actualWork , która jest funkcją. Zwróć uwagę na brak nawiasów w instrukcji return — nie ma ich dlatego, że zwracamy do funkcji referencję, a nie wynik wywołania tej funkcji.
* Ponieważ kod zaczyna się od var a = , wartość zwracana przez samowywołującą się funkcję jest przypisywana do zmiennej a .

Jeśli chcesz sprawdzić, czy poprawnie rozumiesz omówiony zakres materiału, spróbuj odpowiedzieć na poniższe pytania. Jakie będzie zachowanie napisanego przed chwilą programu, gdy:

* zostanie po raz pierwszy załadowany?
* po załadowaniu zostanie wywołana funkcja a () ?

Domknięcia

Zanim zajmiemy się domknięciami, powtórzmy i rozszerzmy trochę pojęcia zakresu w języku JavaScript.

Łańcuch zakresów

JavaScript nie wyróżnia żadnych zakresów ograniczonych nawiasami klamrowymi, ale istnieje zakres funkcji. Zmienna zdefiniowana wewnątrz funkcjinie jest widoczna poza nią, natomiast zmienna zdefiniowana wewnątrz bloku kodu (np. po if lub w pętli for) jest dostępna poza blokiem.

var global = 11;

function foo() {

  var local = 11;

*return* global;

}

console.log(foo()); *// 11*

console.log(local); *// ReferenceError: local is not defined*

Zmienna global ma zakres globalny, natomiast zmienna local zakres funkcji foo(). Zatem:

* Wewnątrz foo() widoczne są zarówno global jak i local
* Na zewnątrz foo() widoczna jest zmienna global, ale nie zmienna local

Jeśli zdefiniujemy funkcję inner () zagnieżdżoną w outer () , będzie ona miała dostęp do zmiennych ze swojego zakresu, a także do zmiennych swoich funkcji nadrzędnych. W takim wypadku mówimy o łańcuchu zakresów, który może być dowolnie długi (głęboki).

var global = 1;

function outer() {

  var outer\_local = 2;

  function inner() {

    var inner\_local = 3;

*return* outer\_local + inner\_local + global;

  }

*return* inner();

}

console.log(outer()); *// 6; funkcja inner ma dostęp do wszystkich zmiennych*

Przerwanie łańcucha za pomocą domknięcia

var a = 'zmienna globalna';

var F = function () {

  var b = 'zmienna lokalna';

  var N = function () {

    var c = 'wewnętrzna lokalna';

  };

};

*// Gdy N wydostanie z zakresu F i trafi do przestrzeni globalnej*

*// będziemy mieli do czynienia z domknięciem.*

*// Funkcja N zamknie swój zakres i zabierze go do przestrzeni globalnej.*

*//*



Funkcje posiadają własną przestrzeń, którą mogą wykorzystywać do przechowywania innych zmiennych, takich jak b , oraz funkcji wewnętrznych, takich jak N .



Jeśli jesteś w punkcie a, jesteś w przestrzeni globalnej. Jeśli w punkcie b , który należy do przestrzeni funkcji F , masz dostęp do przestrzeni globalnej oraz do przestrzeni F . Jeśli znalazłeś się w punkcie c , który należy do funkcji N , masz dostęp do przestrzeni globalnej, przestrzeni F oraz N . Nie da się sięgnąć z a do b , ponieważ punkt b nie jest widoczny poza F . Można natomiast uzyskać dostęp z c do b lub z N do b . Interesujące jest to, że efekt domknięcia ma miejsce, gdy jakimś sposobem N wydostaje się z F i trafia do przestrzeni globalnej.



Co się wtedy dzieje? N jest w tej samej przestrzeni globalnej co a . Jako że funkcje pamiętają środowisko, w którym zostały zdefiniowane, N nadal ma dostęp do przestrzeni F , a co za tym idzie, dostęp do b . Jest to ciekawe dlatego, że N znajduje się tam gdzie a i nadal ma dostęp do b , natomiast a nie ma dostępu do b .

Jak N udaje się przerwać łańcuch? Istnieją dwa sposoby: N może zostać zmienną globalną (pominięcie var ) lub może zostać zwrócona ( return ) przez F do przestrzeni globalnej. Zobaczmy, jak to wygląda w praktyce.

Domknięcie 1.

Zmodyfikujmy funkcję F z powyższego przykładu, niech F zwraca N , a N zwraca b i ma do niej dostęp poprzez łańcuch zakresów:

var a = 'zmienna globalna';

var F = function () {

  var b = 'zmienna lokalna';

  var N = function () {

    var c = 'wewnętrzna lokalna';

*return* b;

  }

*return* N;

}

var inner = F();

console.log(inner()); *// Funkcja globalna inner ma dostęp*

*// do prywatnej przestrzeni F*

Ponieważ F () można wywołać z przestrzeni globalnej (jest funkcją globalną), możesz ją wywołać i przypisać zwracaną przez nią wartość do innej zmiennej globalnej. Wynikiem będzie nowa funkcja globalna, która ma dostęp do prywatnej przestrzeni F ()

var inner = F();

console.log(inner()); // “zmienna lokalna”

Domknięcie 2.

Powyższy wynik można uzyskać nie co inaczej. Funkcja F() zamiast zwracać funkcję utworzy w swoim ciele nową globalną funkcję inner()

var inner; *// element zastępczy*

var F = function () {

  var b = 'zmienna lokalna';

  var N = function () {

*return* b;

  }

  inner = N;

};

F();

console.log(inner()); *// zmienna lokalna*

Wewnątrz przestrzeni F() definiowana jest funkcja N(), która ma dostęp do jej zakresu i której referencja jest przypisana do zmiennej globalnej inner, zatem funkcja inner() będzie miała dostęp do zakresu funkcji F() mimo iż jest częścią przestrzeni globalnej.hhhhhhh

Domknięcie 3. i definicja

Każda funkcja może być uznana za domknięcie gdyż każda funkcja utrzymuje tajne powiązanie ze środowiskiem (zakresem), w którym została utworzona. Jednak w większości przypadków ten zakres jest niszczony, jednakże jak pokażują powyższe przykłady zakres ten może zostać utrzymany.

Domknięcie jest tworzone, gdy funkcja zachowuje powiązanie z zakresem funkcji nadrzędnej nawet po tym, jak funkcja nadrzędna zakończyła działanie.

Parametry funkcji zachowują się jak zmienne lokalne dla tej funkcji, ale są tworzone domyślnie. Nie trzeba używać dla nich słowa kluczowego var . Można utworzyć funkcję zwracającą inną funkcję, która z kolei zwraca parametr swojej funkcji nadrzędnej.

function foo(param) {

  var noo = function () {

*return* param;

  };

  param++;

*return* noo;

}

var inner = foo(123);

console.log(inner()); *// 124*

Zmienna param została zwiększona już po definicji funkcji, a pomimo to inner() zwróciła aktualną wartość. Jest to dowód na to, że funkcja utrzymuje referencję do zakresu,( a nie do zmiennych i ich wartości znalezionych w zakresie podczas wykonywania funkcji. ???)

Zmodyfikujmy troche powyższy kod:

function foo(param) {

  var noo = function () {

*return* param;

  };

  param++;

  console.log(noo());

*return* noo;

}

console.log(foo(123));

*// 124*

*// [Function: noo]*

Domknięcia w pętli

Przyjrzyjmy się teraz kanonicznemu błędowi nowicjuszy w kwestiach związanych z domknięciami. Często prowadzi on do bardzo trudnych do wykrycia błędów, ponieważ na pierwszy rzut oka wszystko wygląda normalnie.

function foo() {

  var arr = [], i;

*for* (i = 0; i < 3; i++) {

    arr[i] = function () {

*return* i;

    };

    console.log(arr[i]());

  }

*return* arr;

}

var arrGlobal = foo();

*for* (var i = 0; i < arrGlobal.length; i++) {

  console.log(arrGlobal[i]());

}

*// 0*

*// 1*

*// 2*

*// 3*

*// 3*

*// 3*

Funkcje nie pamiętają wartości, tylko przechowują referencję do środowiska, w którym zostały utworzone. W tym przypadku zmienna i rezyduje akurat w środowisku, w którym zostały zdefiniowane te trzy funkcje. Dlatego wszystkie funkcje sięgają do tego środowiska i znajdują najbardziej aktualną wartość i . Po wyjściu z pętli wartością zmiennej i jest 3 . Wszystkie trzy funkcje wskazują więc na tę samą wartość.

Eleganckie rozwiązanie polega na wykorzystaniu kolejnego domknięcia, tak jak pokazano poniżej:

function foo() {

  var arr = [], i;

*for* (i = 0; i < 3; i++) {

    arr[i] = (function (x) {

*return* function () {

*return* x;

      };

    })(i);

  }

*return* arr;

}

var arrGlobal = foo();

*for* (let i = 0; i < arrGlobal.length; i++) {

  console.log(arrGlobal[i]());

}

*// 0*

*// 1*

*// 2*

Bieżąca wartość i jest przekazana do innej funkcji natychmiastowej, w której staje się zmienną lokalną x, która za każdym razem ma inną wartość.

Ten sam wynik uzyskamy wykorzystująć funkcję wewnętrzną:

function foo() {

  function binder(x) {

*return* function () {

*return* x;

    };

  }

  var arr = [], i;

*for* (i = 0; i < 3; i++) {

    arr[i] = binder(i);

  }

*return* arr;

}

var arrGlobal = foo();

*for* (i = 0; i < arrGlobal.length; i++) {

  console.log(arrGlobal[i]());

}

*// 0*

*// 1*

*// 2*

Kluczem do sukcesu jest wykorzystanie funkcji pośredniej do uczynienia wartości i lokalną podczas każdej iteracji

Funkcje dostępowe

Kolejnym przykładem użycia domknięcia jest utworzenie funkcji dostępowych: pobierającej (ang. getter ) i ustawiającej (ang. setter ). Funkcja ustawiająca wartość może zawierać pewną logikę do walidacji wartości przed przypisaniem jej do chronionej zmiennej.

Obie funkcje dostępowe można umieścić wewnątrz tej samej funkcji, która zawiera zmienną secret , tak by współdzieliły ten sam zakres:

var getValue, setValue;

(function () {

  var secret = 0;

  getValue = function () {

    return secret;

  };

  setValue = function (v) {

    if (typeof v === "number") {

      secret = v;

    }

  };

}());

console.log(getValue());

setValue(321);

console.log(getValue());

setValue(false);

console.log(getValue());

W tym przypadku funkcja zawierająca wszystkie elementy jest funkcją natychmiastową. Definiuje ona setValue () i getValue () jako funkcje globalne, podczas gdy zmienna secret pozostaje lokalna i nie jest dostępna bezpośrednio.

Iterator

Poniższy przykład pokazuje wykorzystanie domknięcia w celu osiągnięcia funkcjonalności iteratora.

Oto funkcja inicjująca, która przyjmuje tablicę wejściową, a także definiuje prywatny wskaźnik i , zawsze wskazujący następny element w tablicy:

// Wykorzystanie domknięcia jako iteratora

function setup(x) {

  var i = 0;

  return function () {

    return x[i++];

  };

}

var next = setup(['a', 'b', 'c', 'd']);

console.log(next());

console.log(next());

console.log(next());

console.log(next());

IIFE a bloki

Ponieważ specyfikacja ES5 nie zapewniła zakresu bloku, popularnym wzorcem dla uzyskania zakresu bloku było użycie natychmiastowo wywoływanego wyrażenia funkcyjnego (ang. immediately invoked function expressions — IIFE), np.:

(function () { var block\_scoped=0; }()); console.log(block\_scoped); // ReferenceError: block\_scoped is not defined

Dzięki obsłudze zakresów bloków przez ES6 można po prostu użyć deklaracji let lub const.

Funkcje strzałkowe

W JavaScripcie zawsze pisaliśmy wyrażenia funkcyjne. Idiomatyczne jest pisanie w JavaScripcie kodu takiego jak ten (przykład w jQuery):

$("#submit-btn").click(function (event) {

validateForm();

submitMessage();

});

Ten styl pisania wyrażeń funkcji anonimowych jest znany jako **funkcje lambda** . Tę funkcjonalność obsługuje kilka innych języków. Chociaż funkcje lambda są mniej lub bardziej standardowe w nowych językach, ich użycie spopularyzował JavaScript. Jednak składnia lambda w JavaScripcie nie była zbyt zwięzła. Funkcje strzałkowe ES6 wypełniają tę lukę i zapewniają zwięzłą składnię dla pisania funkcji.

const num = [1,2,3]

const squares = num.map(function(n){

return n\*n;

});

console.log(squares); // [1,4,9]

Powyższy kod można uprościć stosując składnię funkcji strzałkowej:

const squaresTwo = num.map (n => n \* n)

Gdy potrzebujemy wielu argumentów, musimy zawrzeć listę argumentów w nawiasach okrągłych:

* Brak parametrów : () => {...} ;
* Jeden parametr : a => {...} ;
* Więcej niż jeden parametr: (a, b) => {...}

Funkcje strzałkowe mogą zawierać zarówno ciała instrukcji (ang. statement bodies ), jak i ciała wyrażeń (ang. expression bodies ):

n => { return n+n} // blok instrukcji

n => n+n // wyrażenie

Oba zapisy są równoważne, ale druga odmiana jest zwięzła i preferowana. Funkcje strzałkowe są zawsze anonimowe. Jednym z ważnych aspektów funkcji strzałkowych, jest to, że funkcje strzałkowe nie wiążą własnych wartości słowa kluczowego this — wartość jest leksykalnie wywodzona z otaczającego zakresu.

Obiekty

Od tablic do obiektów

Tablica w JS jest listą wartości. Ponieważ każdy element tablicy ma swój index i wartość to tablicę możemy traktować jako uporzątkowany zbór par klucz-warość. Kluczem jest oczywiście liczbowy index zaczynający się od zera.

Definując obiekty stosujemy klucze, które są nazwami i są oddzielone od wartości dwukropkiem. Elementy obiektu (zwane właściwościami lub polami) oddziela się przecinkami. Definicję obiektu ujmujemy w nawiasy klamrowe i przypisujemy stałej.

Przykład

const student = {

name: ‘Piotr’,

surname: ‘Piekarz’,

info: function(){

console.log(`Jestem studentem, nazywam się ${this.name} ${this.surname}`);

}

};

Nie zaleca sie stosowania cudzysłowów(oszczędność znaków), jednakże jeśli chcemy właściwości nadać nazwę, która nie jest poprawną nazwą zmiennej należy je zastosować.

Tablice zdefiniowane za pomocą [ ] – **literały tablicowe,** a obiekty za pomocą { } – **literały obiektowe**

Elementy, właściwości, metody i składowe

Tablice zawierają elementy, zaś obiekty właściwości.

Właściwość obiektu może wskazywać funkcję bowiem funkcja jest daną, wtedy nazywamy ją **metodą.** Ogólnie właściwości nazywamy **składowymi**.

Tablice asocjacyjne

W niektórych językach programowania istnieje podział na :

* tablice indeksowe – kluczami są liczby
* tablice asocjacyjne, zwane inaczej tablicami mieszającymi lub słownikami – kluczami są łańcuchy znaków lub dowolne obiekty

W JS tablicą indeksowym odpowiadają tablice, a tablicą asocjacyjnym – obiekty.

Dostęp do właściwości obiektu

* przy użyciu nawiasów kwadratowych – student[‘name’]
* przy użyciu kropki – student.surname

Gdy będziem się odwoływać do nieistniejącej właściwości to otrzymamy wartość undefined.

Obiekty mogą zawierać dane, w tym inne obiekty:

const book = {

  name: 'Paragraf 22',

  published: 1961,

  author: {

    firstName: 'Joseph',

    lastName: 'Heller'

  }

};

console.log(book['author']['firstName']);

console.log(book.author.lastName);

console.log(book.author['firstName']);

let key = 'name';

console.log(book[key]);

Wywoływanie metod obiektu

Dostęp do metod odbywa się tak jak do zwykłych właściwości: przy użyciu kropki lub notacji nawiasowej. Metody wywołujemy dodając nawiasy.

Modyfikacja właściwości i metod

Składowe(właściwości i metody) istniejących obiektów można modyfikować w dowolnym momencie. Można dodawać nowe i usuwać stare. Można utworzyć pusty obiekt, a składowe dodawać póżniej.

*// Deklaracja pustego obiektu*

const student = {};

student.name = 'Piotr';

student.surname = 'Piekarz';

student.sayName = function(){

  console.log(`Jestem studentem i mam na imię ${student.name}`);

}

student.sayName();

Wartość this

Specjalna wartość this umożliwia dostęp z wnętrza metody do aktualnego obiektu ( tzn do obiektu, do którego należy metoda).

student.saySurname = function(){

  console.log(`Jestem studentem i nazywam się ${this.surname}`);

}

student.saySurname();

Konstruktory

Konstruktor jest to funkcja za pomocą, której można tworzyć obiekty, może ona przyjmować parametry.

function Point(x, y){

  this.coordinateX = x;

  this.coordinateY = y;

  this.module = function(){

*return* Math.sqrt(

      Math.pow(this.coordinateX, 2) + Math.pow(this.coordinateY, 2)

    ).toFixed(2);

  }

}

const pointA = new Point(3, 4);

const pointB = new Point(4, 3);

console.log(pointA.module()); // 5

console.log(pointB.module()); // 5

Obiekt globalny

Programy napisane w JS są uruchamiane wewnątrz środowiska (np. przeglądarki). Środowisko zapewnia obiekt globalny, a wszystkie zmienne globalne są jego właściwościami.

W środowisku przeglądarki obiektem globalnym jest **window.** Można do niego uzyskać dostęp poprzez użycie słowa kluczowego **this** poza konstruktorem np. w globalnym kodzie programu poza jakąkolwiek funkcją. Wszystkie własności ustawione za pomocą this stają się własnościami obiektu globalnego.

Jeśli zadeklarujemy zmienną globalną

var a = 11;

to będziemy mogli się do niej odwołać na różne sposoby:

console.log(a); *// 11*

console.log(window.a); *// 11*

console.log(window['a']); *// 11*

console.log(this.a); *// 11*

Zadeklarowanie konstruktora i wywołanie go bez new zwróci undefined

function Student(name) {

  this.name = name;

}

var mathematicsStudent = Student('Jan Kowalski');

console.log(typeof mathematicsStudent); *// undefined*

console.log(typeof mathematicsStudent.name); *// Uncaught TypeError: Cannot read p// roperty 'name' of undefined*

Ponieważ w funkcji Student pojawiło się słowo kluczowe this, utworzona została zmienna globalna ( właściwość obiektu globalnego) o nazwie name.

console.log(name); *// 'Jan Kowalski'*

console.log(window.name); *// "Jan Kowalski"*

Jeśli konstruktor zostanie wywołany z użyciem słowa new, zwrócony zostanie nowy obiekt, do którego będzie się odnosiło słowo this.

var studentOfPhilosophy = new Student('Amadeus Socrates');

console.log(typeof studentOfPhilosophy); *//"object"*

console.log(studentOfPhilosophy.name); *// "Amadeus Socrates"*

Także wbudowane funkcje globalne można wywołać jako metody obiektu window.

console.log(parseInt('7 cudów świata')); *// 7*

console.log(window.parseInt('7 cudów świata')); *// 7*

Właściwość construktor

Gdy obiekt jest tworzony otrzymuje właściwość o nazwie construktor. Zawiera ona referencję do konstruktora, za pomocą, którego został on utworzony.

function Point(x, y){

  this.coordinateX = x;

  this.coordinateY = y;

  this.module = function (){

*return* Math.sqrt(

      Math.pow(this.coordinateX, 2) + Math.pow(this.coordinateY, 2)

    ).toFixed(2);

  }

}

const pointA = new Point(3, 4);

const pointB = new Point(4, 3);

console.log(pointA.module());

console.log(pointB.module());

Za pomocą tej właściwości można utworzyć nowe obiekty:

const pointC = new pointA.constructor(2, 3);

console.log(pointC.module());

Konstruktorem obiektów literałowych jest wbudowana funkcja Object().

const point = {

  coordinateX: 3,

  coordinateY: 4,

  determinatePoint(x, y) {

    this.coordinateX = x;

    this.coordinateY = y;

  }

}

console.log(point.constructor); // ƒ Object() { [native code] }

Operator instanceof

Operator ten sprawdza, czy dany obiekt został utworzony za pomocą określonego konstruktora:

function Student() { }

var philologyStudent = new Student();

var literalObject = {};

philologyStudent instanceof Student; *// true; bez nawiasów odwołujemy się do*

philologyStudent instanceof Object; *// true; referencji a nie do funkcji*

literalObject instanceof Object; *// true*

Funkcje zwracające obiekty

O to funkcja produkująca obiekty:

function factory(coordinateX, coordinateY) {

*return* {

    coordinateX: coordinateX,

    coordinateY: coordinateY

  };

}

const pointA = factory(3, 7);

console.log(pointA); // {coordinateX: 3, coordinateY: 7}

console.log(typeof pointA); // “object”

Tak są tworzone obiekty wewnątzr konstruktorów:

function Constructor() {

*// var this = {}; to jest pseudokod*

  this.a = 11;

*// return this; zwracanie this*

}

Obiekty wbudowane

Function

Funkcje są specjalnym typem danych ale także są obiektami. Istnieje wbudowany konstruktor Function( ), za pomocą którego możemy tworzyć funkcje, choć nie jest to zalecany sposób tworzenia funkcji, gdyż posiada on wszystkie wady funkcji eval( ).

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

console.log(sum(3, 4)); *// 7*

Powyższy kod jest równoważny dwom następującym sposobom tworzenia funkcji:

*// Pierwszy sposób*

var sumOne = function (a, b) {

*return* a + b;

};

console.log(sumOne(3, 4));

*// Drugi sposób*

function sumTwo(a, b) {

*return* a + b;

}

console.log(sumTwo(3, 4));

Wiele argumentów można zapisać wewnątrz jedego łańcucha znaków:

var first = new Function('a, b, c, d, e', 'return arguments;');

var result = first(2, 3, 4, 5, 7);

console.log(result); *// [Arguments] { '0': 2, '1': 3, '2': 4, '3': 5, '4': 7 }*

Należy unikać funkcji, które jako argument pobierają kod w postaci łańcucha znaków (np. setTimeout czy eval)

Właściwości obiektu Function

Jak wszystkie obiekty, funkcje posiadają właściwość construktor zawierającą referencję do konstruktora Function ( ). Posiadają też właściwość lenght, która określa liczbę parametrów.

Korzystanie z właściwości prototype

Najczęściej wykorzystywaną właściwością funkcji jest **prototype**:

* Właściwość **prototype** obiektu **function** wskazuje inny obiekt
* Ma ona znaczenie tylko, jeśli funkcja jest wywoływana jako konstruktor
* Wszystkie obiekty utworzone za pomocą funkcji przechowują referencję do właściwości **prototype** i mogą korzystać z jej właściwości jak z własnych.

var ninja = {

  name: 'Ninja',

  say: function () {

*return* `Jestem ${this.name}`;

  }

}

console.log(`Prototyp literału obiektowego: ${ninja.prototype}`);

*// Prototyp literału obiektowego: undefined*

console.log(`Konstruktor literału obiektowego: ${ninja.constructor}`);

*//Konstruktor literału obiektowego: function Object() { [native code] }*

console.log(`

Prototyp konstruktora literału obiektowego: ${ninja.constructor.prototype}

`);

*// Prototyp konstruktora literału obiektowego: [object Object]*

console.log(ninja.constructor.prototype); *// {}*

function Foo() { };

console.log(typeof Foo.prototype); *// object*

Foo.prototype = ninja;

var baby\_ninja = new Foo();

console.log(baby\_ninja.name); *// Ninja*

console.log(baby\_ninja.say()); *// Jestem Ninja*

Metody obiektu Function

Funkcje są obiektami i jako takie są potomkami obiektu Object, dlatego też posiadają metody domyślne takie jak toString( ); Jeśli tę metodę wywołamy na funkcji to zwróci jej kod żródłowy ale jeśli na funkcji wbudowanej to zwróci łańcuch znaków [native code] czyli metoda ta może posłużyć do odrażniania metod natywnych od tych zdefiniowanych przez programistów.

function myFunc(a, b, c) {

*return* (a + b + c) / 3;

}

console.log(myFunc.toString());

*// “function myFunc(a, b, c) {*

*//   return (a + b + c) / 3;*

*// }”*

console.log(parseInt.toString());

*//"function parseInt() { [native code] }"*

Metody call i apply

Metod obiektów funkcyjnych call() oraz apply() można użyć do wywołania funkcji i przekazania do nich dowolnych argumentów.

Dzięki nim obiekty mogą wypożyczać metody od innch obiektów i wywoływać je jak własne. (Wielokrotne wykorzystanie kodu).

const some\_obj = {

  name: 'Ninja',

  say: function (who) {

*return* `Sie ma ${who}, jestem ${this.name}`;

  },

someMethod: function (a, b, c) {

*return* a \* b / c;

  }

}

console.log(some\_obj.say('Johny'));

*// Sie ma Johny, jestem Ninja*

const my\_obj = { name: 'Guru programowania' };

console.log(some\_obj.say.call(my\_obj, 'Johny'));

*// Sie ma Johny, jestem Guru programowania*

console.log(some\_obj.someMethod.call(my\_obj, 2, 4, 6));

*// 1.3333333333333333*

Nowe spojrzenie na obiekt arguments

Zmienna arguments, przechowuje wartości wszystkich parametrów użytych podczas wywołania funkcji:

function foo() {

*return* arguments;

}

console.log(foo(1, 2, 3, 4));

*// Arguments(4) [1, 2, 3, 4, callee: ƒ, Symbol(Symbol.iterator): ƒ]*

*// 0: 1*

*// 1: 2*

*// 2: 3*

*// 3: 4*

*// callee: ƒ foo()*

*// length: 4*

*// Symbol(Symbol.iterator): ƒ values()*

*// \_\_proto\_\_: Object*

../../node/node arguments.js

[Arguments] { '0': 1, '1': 2, '2': 3, '3': 4 }

arguments wygląda jak tablica, jednakże jest to obiekt tablicopodobny, pomimo iż posiada indeksowane elementy. Posiada także właściwość length, ale nie posiada metod tablicowych, takich jak sort() czy slice().

function fooOne() {

*// const args = [].slice.call(arguments)*

const args = Array.prototype.slice.call(arguments);

*return* args.reverse();

}

console.log(fooOne(1, 2, 3, 4)); *// [ 4, 3, 2, 1 ]*

Jak wynika z powyższego kodu można wypożyć metodę slice().

Leksykalne this w funkcjach strzałkowych

Funkcje strzałkowe zachowują się inaczej niż zwykłe funkcje, nie posaidają własnej wartości this, tylko dziedziczoną z zakresu otaczającego (leksykalnego).

Funkcje mają specjalną zmienną this, która odnosi się do obiektu, za pomocą którego (metoda) została wywołana. Wartość this jest nadawana dynamicznie na podstawie wywołania funkcji (dynamiczne this).

Funkcja jest wykonywana w dwóch zakresach – leksykalnym i dynamicznym. Zakres leksykalny jest zakresem otaczającym zakres funkcji, zakresem dynamicznym jest zakres, który wywołał funkcję (zazwyczas obiekt).

Prototypy

Właściwość prototype

Dodawanie metod I właściwości przy użyciu prototypu

Konstruktorem nowych obiektów może być stworzona przez nas funkcja. Wewnątrz takiej funkcji wywołanej za pomocą operatora new, będziemy mieli dostęp, poprzez zmienną this, do obiektu, który zostanie zwrócony przez konstruktor.

Ekosystem JavaScript – Strefa kursów

Szybkie wprowadzenie do JavaScript

Co to jest JavaScript?

* JavaScript to język interpretowanay przez przeglądarkę
* Do uruchomiania programów JS potrzebna jest przeglądarka albo inne środowisko np. Node.js
* Umożliwia programowanie zarówno w stylu funkcyjnym jak i obiektowym

Zmienne

Element programu, który przechowuje wartości

var a = 1;

Różne wartości zmiennych:

var a = 11;

var name = ‘Jan’;

var selected = true;

var a = 10;

var b = 34;

var c = 4;

var firstName = "Ksawery";

alert(a + b);

Funkcje

function myFunction(a, b) {

  var result = a + b;

  return result;

}

var x = myFunction(2, 10);

alert(x);

var y = myFunction(3, 5);

alert(y);

Instrukcje warunkowe

var price = 145;

if (price > 200) {

  alert("Cena wyższa niż 200. Otrzymujesz rabat");

} else {

  alert("Cena jest równa 200 lub mniejsza. Nie otrzymujesz rabatu");

}

Iteracja

var i = 1;

while( i < 10){

alert(i);

i++;

}

for ( var i = 1; i <= 10; i++ ) {

    alert(i);

}

Tablice

var numbers = [1, 3, 4, 44, 55, 66];

Tablice posiadają index liczony o zera

var myArray = [ 1, true, ‘finish’, 44];

alert(myArray);

alert(myArray[2];

var myArray = [ 20, 34, "ok", true, 102, "test" ];

for ( var i = 0; i < myArray.length; i++ ) {

    alert(myArray[i]);

}

Stringi