W JS aby wyświetlić wartość variable używa się nastepujacej funkcji:

* Console.log()

Aby stworzyć variable stosuje się:

* Let variable = ...

Możemy stworzyć również variable, którego wartość nie będzie mogła ulec zmianie, stosując Const.

* Const PI = 3,14

Jesli chcemy zrobić komentarz na wiele lini to stosuje się:

/\* tutaj wpisz tekst \*/

Aby sprawdzić typ variable można to zrobić przy pomocy typeof:

Console.log(typeof variable)

Istotna strona gdzie możemy sprawdzić kolejność egzekowwania operatorów:

Mdn operator precedense >

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_precedence>

Aby dodać do anszego string dynamicznie wartości wykorzystujemy do tego ``:

Let test = 8

Console.log(`This is ${test}`)

Dodatkowo tworzenie stringów przy pomocy `` pozwala na łatw tworzenie nowych linijek tekstu.

`test

test`

Warunek if else, stynatx: if (warunek typu coś > od czegoś) { baza, która zadziała jesli warunek zostął spełniony} else { }

Istnieje również else if block aka elif z pythona:

If (michal) {

} else if (ada) {

} else {}

Jeśli warunek if ma tylko jedną część to nie musimy tworzyć {} np.:

If (age === 18) console.log(`This is true`)

Jeśli chcemy manualnie zmienić typ wartości aka type conversion:

Aby uzyskać string: String(x)

Aby uzyskać liczbę: Number(x)

Javascript automatycznie zamienia wartości aka type cohersion np:

`23` - `10` - 3 da nam liczbę 10, ale jesli zrobimy: `23` + `10` + 3 to dostaniemy string 2313

Truthy and Falsy values

Istnieje 5 nieprawdziwych wartości: 0, ‘’, undefined, null, NaN (not a number).

W przypadku gdybyśmy chcieli sprawdzić przy pomocy warunku if czy variable jest True, a posiadałby jedną z powyższych wartości to dsotalibyśmy False.

**Equality Operators == i ===.**

Potrójny === nazywa się strict equality operator i spełniony ejst tylko jeśli obywdwie przyrównywane wartości sa dokładnie takie same.

Podwójny == nazywa się lose equality operator i np gdy przyrwónujemy „18” == 18 dostaniemy True. Zasadniczo staramy się unikać tego.

Uzyskanie wartości od użytkownika aka input, w tym celu używa się funkcji **prompt**:

const user\_value = prompt(„what’s your favorite color”)

Boolean operators = and, or, not(!) < tak jak w Python.

And operator jest zapisywany jako &&,

Or operator jest zapisywany jako ||,

Not operator jest zapisany jako !n,

**Switch statments:**

Działają na zasadnie bloków if ale z inną kosntrukcją np.:

Switch(day) {

case ‘monday’:

consolo.log(„…”)

consolo.log(„…”)

**break**

case ‘tuseday’:

consol.log(„…”)

**break**

case ‘wednesday:

case ‘thursday: (zadziałą dla obydwu rzeczy)

consol.log(„…”)

**break (jest potrzebny bo inaczej kod przejdzie do dalszej częsci)**

default:

consol.log(“Not a valid day”)

}

**Statements and Expressions**

**Expressions to np. 3 +4, 1991, true && false && !false**

**Statements to większa część kodu np. if else statements.**

**Conditional (Ternary) Operator**

Poniżej jest budowa conditional operatora, gdzie tworzymy warunek, następnie znakiem ? tworzymy blok spełnionego założenia i po : blok negatywnego spełnienia. Pozwala to zaoszczędzić dużo czasu przy prostych sprawdzeniach warunku if else.

Const age = 24

age >= 18 ? console.log(“I like alcohol”) : console.log(„I like wataa”)

Z przypisaniem do variable:

Const drink = age >= 18 ? “wine”: “water”

**Strict Mode**

Celem strict mode jest umożliwienie włączenie widzenia błędów oraz uniemożliwia wykonanie pewnych funkcji.

Aby aktywować strict mode należy dodać na początku skryptów **„use strict”**

**Funkcje**

Funkcje tak jak w python, to blok kodu, który można w łatwy sposób ponownie wykorzystać. Konstrukcja:

Function Blob(parameter1, parameter2) {

Console.log(parameter1, parameter2)

}

**Functions Declarations vs Expressions**

Obydwie formy pisania funkcji są właściwe, osobiście preferuje f. declaration. Różnica między nimi polega, że declaration f. może być wezwana przed jej blokiem.

Function declaraction:

function calcAge1 (birthYear) {

return 2033 - birthYear

}

const age1 = calcAge1(1991)

console.log(age1)

Function expression:

const calcAge2 = **function** (birthYear) {

return 2037 - birthYear

}

const age2 = calcAge2(1991)

console.log(age1)

**Arrow Function**

Jest to forma funkcji o bardzo prostej budowie, nadająca się do prostych funkcji, która automatycznie zwraca wartość swojego ciała.

Const calcAge3 = birthYear => 2033 – birthyear

Console.log(calcAge3(1990))

A co w przypadku bardziej rozbudowanej funkcji:

Const yearsUntilRetirement = birthYear => {

const age = 2037 – birthYear

const retirement = 65 - age

return retirement

}

W przypadku wielu parametrów:

Const yearsUntilRetirement = (birthyear, firstName) => {

const age = 2037 – birthYear

const retirement = 65 - age

return `${firstName} retirement in ${ retirement }`

}

**Functions Calling Other Functions**

Dla mnie logiczne.

Const cutFrutPieces = function (fruit) {

return fruit \* 4

}

Function fruitProcessor(apples, oranges) {

Const applePieces = cutFrutPieces(apples)

Const orangePieces = cutFrutPieces(oranges)

const juice = `Juice with ${applePieces} apples and ${orangePieces} oranges`

return juice

}

**Arrays**

Tak jak w Python syntax: const lista = [x, y, z]

Jest też drugi sposób: const lista = new Array(x, y, z)

Aby uzyskać długość listy stosuje się jej paramter: **lista.length**

Const Listy moga być zmieniane bo nie należa do primitive values, jak variable. Jednak nie mogą być zmienione w pełni. Np poniższa zamiana zwróci nam error:

Const lista = [x, y, z]

Lista = [b, c]

Arrays Methods:

PUSH – funkcja, która dodaje(mutuje orginalną listę) na koniec listy element:

lista.push(a)

ciekawostka, push zwraca nową długość zmutowanej listy:

const newlength = lista.push(a) -> zwróci 4.

UNSHIFT – funkcja, która dodaje element na poczatek listy:

lista.unshift(a)

POP – funkcja, która zabiera ostatni element z listy:

Lista.pop()

Tak jak w Python, pop zwraca element zabrany z listy:

Const element = Lista.pop() - > zwróci nam „z”

SHIFT – funkcja, która zabiera z początku listy element:

Lista.shift()

Tak jak w Python, pop zwraca element zabrany z listy:

Const element = Lista.shift() - > zwróci nam „x”

INDEXOF – funkcja, która zwraca nam index elementu w danej liście:

Lista.indexOf(y) -> zwróci nam 1.

Jeśli nie ma danego elemetu w liście zostanie nam zwrócone -1.

INCLUDES – funkcja, która zwraca nam true or false jeśli element znajduje się w danej liście.

Lista.includes(abc) -> zwróci nam False

**Introduction to objects**

**OBJECTS –** Obiekty sa podobne w konstrukcji do struktóry dictionary w Python. W JS mówi się, że obiket michal ma property name o wartości „Michał”.

Const michal = {

Name: „Michał”,

Lastname: „Krepiniewicz”,

Etc..

}

Dot vs. Bracket Notation

Tak jak w python, można uzyskac property poprzez dostanie się do obiektu na dwa sposoby:

Obiekt.name lub obiekt[„name”]

Zasadniczo używa się dot notation aby uzyskać wartość z obiektu ale tylko w przypadku kiedy znamy property name. Gdybyśmy chcili skorzystać z prmp, gdzie zadajemy pytanie i opcje do wyboru jak: age, name etc. To nalezy skorzystać z brackets:

Const data = prompt(„what do you ..., chose from age, firstname etc. ”)

naszObiekt[data]

gdybyśmy zrobili to z wykorzystaniem . naszObiekt.data dostalibyśmy undeifned bo nie istenieje taka właściwość obiektu.

Aby dodac elementy do obiektu możemy to zrobić na 2 sposoby:

naszObiekt[„location”] = „Warsaw”

albo

naszObiekt.facebook = „Fuck You”.

Ciekawa opcja do wyświetlenia danych obiektu w formie tabeli jest skorzystanie z **console.table(obiekt);**

**Objects Methods**

Obiekty moga posiadać swoje własne funkcje, np.

michal = {

birthYear: 1991,

calcAge: function (birthYear) { return 2037 – birthYear }

}

michal.function(1991) lub michal[“calcAge”](1991)

Aby dostac się property obiektu stosuje „**this**” variable/object, który równa się naszemu obiektowi ala self z Pythona.

michal = {

birthYear: 1991,

calcAge: function () { return 2037 – this. birthYear }

}

michal.function() lub michal[“calcAge”]()

Możemy również stworzyć nowe property przy pomocy wbudowanej funkcji:

michal = {

birthYear: 1991,

calcAge: function () {

this.age = 2037 – this. birthyear

}

}

michal.function() lub michal[“calcAge”]()

**For Loop**

Zasada działania jest jak w pytonie, jednak syntax jest inny:

Wzywamy for, gdzie w body mamy 3 wyrażenia: 1) które tworzy zmienną, 2) warunek

sprawdzający, 3) wpłynięcie na stworzoną zmienną na koniec loopa.

for( 1. let rep = 1; 2. rep&lt;11; 2. rep++) {

    console.log(rep)

}

const types = [];

for(i=0; i&lt;=mikeArray.length-1; i++) {

    // One of the ways of adding things into arrays

    types.push(typeof(mikeArray[i]))

    // Second way of adding things into arrays

    types[i] = typeof mikeArray[i]

    // Reading of the array

    console.log(mikeArray[i], typeof(mikeArray[i]))

}

console.log(types)

Mamy również continue and break statement. Continue sprawia, że loop przeksakuje do

kolejnej pętli. Break sprawia, że cały loop ulega przerwaniu.

Loop backwards

// Loop backwards

const mikeArray = [

    &quot;Mike&quot;, &quot;Krepiniewicz&quot;, 2037-1990, &quot;programmer&quot;, [1, 2 ,3]

]

for(let i=mikeArray.length-1; i&gt;=0; i--){

    console.log(mikeArray[i])

}

**While Loop**

Konstrukcja while loop jest dość ciekawa, tworzymy tylko warunek:

While Loop

let rep = 0

while(rep &lt;=10){

console.log(`${rep}`)

rep++

}

Drugi przykład z losowniem losowej liczby aż nie osiągniemy docelowej:

let score;

while(score !== 6){

    score = Math.trunc(Math.random()\*6)+1

    console.log(score)

}

Math.random zwraca nam liczbę pomiędzy 0 a 1, np. 0.2345, 0.4592461 etc.

Math.trunc pozbywa się wartości po przecinku.

Ciekawa rzeczy warunek gdzie: score === 6, nie działa.

**Setting up Prettier in Vs Code**

A screenshot of a computer

Description automatically generated

Następnie po ściagnięciu I zainstalowaniu należy zrobić „Enable” i wskazać go jako default format w settings:

A screenshot of a computer

Description automatically generated

Jesli jakaś bazowa funkcja nie pasuje w Prittier możemy wpłynać na bazowe sutawienia poprzez stworzenie pliku .priettierrc



Snippets

Dzięki snippets możemy skonfigurować takie rzeczy jak uproszczenie pewnych funkcji jak: console.log(). W settings > snippets > new global snipet.

A screenshot of a computer program

Description automatically generated

VS Code extension do Live Server



Nastepnie instalujemy globalnie (-g) live-server:

A screen shot of a computer code

Description automatically generated

Jeśli dostaniemy bład przy odpaleniu live-server należy zmienić politkę uruchomiania skryptów w powershell jako admin:

A screen shot of a computer program

Description automatically generated

Aby skorzystać z live-server wystarczy, że w terminalu wpiszemy: **live-server**

A computer screen with text

Description automatically generated

Debugger przeglądarkowy

Można sprawdzać w przeglądarce w zakładce debugger ewentulane buggi zamiast w środowisku developera:

A computer screen shot of a program

Description automatically generated

Możemy dodać manualnie break point w postaci debuggera w wanyszm kodzie:

A screen shot of a computer program

Description automatically generated

Lesson 5 – Projects

Ważna rzecz, warto jest tworzyć osobne zmienne wewnątrz naszego kodu js zamiast pobierać co chwile dane z elementu html.

Aby wybrać element z dokuentu, stosujemy: document.querySelector(‘’). Wewnątrz wskazujemy element przy pomocy jego classy albo indexu. Sam skorzystanie z quesrySelctor zwróci nam cały element HTML.

**console.log(document.querySelector('.message'))**

Aby uzyskać wartość z danego elementu należy skorzystać z querySelector(„.test”).textContent:

**console.log(document.querySelector(".message").textContent)**

Aby uzyskać wartość z inputu użytkownika stosuje się atrybut .value querySelector(„test”).value:

**document.querySelector(".guess").value = ""**

Jeśli chcemy wpłynąć na atrybut jakim jest style css np. background color stosuje się:

**document.body.style.backgroundColor = „red”**

Jest kilka sposób na kontrolowanie tego co się dzieje w naszym dokumencie HTML aka event control. Jedną z nich jest przypisanie funkcji, do np. przycisku, tak jak ja zdobiłem poniżej, gdzie w elemencie buton dodaliśmy atrybut onclick=”nazwa funkcji”.

A computer screen shot of a program code

Description automatically generated

Inną opcją jest stworzenie evenlistenera w naszym pliku javascript, gdzie podczas kliknięcia jest wzywana stworzona funkcja:

Document.querySelector(„.class”).AddEventListener(„click”, funkcja)

A screen shot of a computer

Description automatically generated

Inną opcją jest też stworzenie event listenera, gdzie wprowadzamy funkcję oraz jej zawartość:

A computer screen shot of text

Description automatically generated

**DOM to document object model** – Strukturowa reprezentacja dokumentu HTML, pozwalająca Javascript na dostęp do elementów HTML i Stylów (CSS) i ich manipulacje. Dom jest tworzony w momencie załadowania strony. Struktura wygląda jak drzewo, gdzie mamy elementy rodziców, dzieci, rodzeństwa.

A diagram of a flowchart

Description automatically generated

Aby wybrać wszystkie lementy, które posiadają tą samą klasę, należy zasotoswać document.querySelectorAll(„.classa”)



Zawartośc jaką uzyskamy dzięki temu przypomina liste z elementami. Posiadając tą liste możemy, wpłynąć na elementy z zapytania w różny sposób np.:

Nadać eventlistern do przycisków

for(i=0; i<btnsOpenModal.lenght; i++) {

btnsOpenModal[i].addEventListener(“click”, function() {

document.body.style.backGroundColor = “black”

})

}

Możemy rownież wpłynąć na classy danego elementu poprzez wykorzystanie attrybytu pobranego elemetu: **classlist.**

A screenshot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

Jesli chodzi o Eventlistener związany z Key mamy 3 rodzaje:

* Keydown - zachodzi w momenecie w którym wciskamy klawisz
* Keypress – zachodzi podczas wielokrotnego wciskania klawisza
* Keyup – zachodzi w momencie w którym puszczamy palce z klawisza

Podczas wcisnięcie klawiszza JS tworzy obiekt, zawierajacy informacje np. Jaki klawisz został wciśnięty. Dzieki temu jesteśmy wstanie np. Stworzyć warunek wykorzystujący parametr.

Np: Parametr „e” jest obiektem stworzonym przez JS w momencie wciśnięcia klawisza.

A screen shot of a computer program

Description automatically generated

Inną formą wpływania na property .classlist jest toggle. Toggle działa jak add/remove, jeśli nie ma takiej właściwości to zostaje ona dodana, jeśli jest zostaje ona usunięta.

A computer code on a black background

Description automatically generated

JavaScript Theory Lecture

A screenshot of a computer screen

Description automatically generated

Właściwości JS:

* High level language: wpływanie na wykorzystanie pamięci komputera na np. stworzenie zmiennej jest zautomatyzowane przez język JS. Jednak problem tego jest taki, że nigdy nie będziemy mieli tak szybkiej apliakcji jak napisanej w języku C.
* Grabage collected: Usuwanie z pamięci komputera niepotrzebnych elementów.
* Interpreted or just-in-time compiled: zautomatyzowane przetowrzenie kodu Js na kod w języku maszynowym.
* Mulit-paradigm language: Podejście do struktury kodu, które wpłynie na style oraz technikę. Zawiera się w tym:
  + Programowanie proceduralne
  + Programowanie obiektowe OOP
  + Programowanie funkcyjne FP
* Prototype-based object-oriented: tworząc obiekty, uzyskują one właściwości od prototypów, np. array uzykuje właściwości i methody od prototyp, który je określa.
* First-class functions: Właściwość ta pozwala rozpatrywać funkcje jako zmienne, które mogą być przekazywane do innych funkcji i zwracane z innych funkcji.
* Dynamic: Typ zmiennej może być zmieniony na inny.
* Single-threaded: JavaScript może wykonywać jedną rzecz na raz. W przypadku kiedy np. korzystamy z zewnętrznego API i oczekujemy na wynik, to nie możemy zablokować naszego single threada. W tym celu stosuje się event loop, który działa w tle.

A screenshot of a computer

Description automatically generated

**JavaScript Engine** - to program komputerowy, którego zadaniem jest wykonuje kod JavaScript. Każda strona posiada JS engine, a najbardziej popularna to **Google’s V-Eight**, który również stosuje się w Node.js.

Każda JS Engine składa się z:

1. **Call Stack** – Miejsce, gdzie nasz kod jest wykonywany przy pomocy **Execution Context**.

Oraz

1. **Heap** – Nieustrukturyzowana pamięć, gdzie przetrzymywane są obiekty, wykorzystywane przez naszą aplikacje.

Jak działa JS Engine:

A diagram of a computer system

Description automatically generated

1. Pierwszym krokiem po przesłaniu kodu do JS Engine jest sczytanie go, określane jako **Parsing**. Zaczytany kod jest w formie struktury danych, określanej mianem Abstract Syntax Tree AST). Każdy element kodu jak funkcje, const etc. są zaczytane i potem uporządkowane w formie AST.
2. Kolejny krok to compilacja przygotowanego AST na kod maszynowy.
3. Następnie przygotowany kod jest od razu wykonywany w Call Stack.
4. W tle przygotowany machine kod przechodzi optymalizacje i kompilacje, aby ponownie zostać wykorzystanym.

Cały ten proces zachodzi w wątkach, do których nasz kod nie ma dostępu.

Rodzaje w jaki sposób kod zostaje przetworzony na kod maszynowy:A diagram of steps

Description automatically generated

**Czym jest Run Time?**

Run Time – Moment, w którym program jest wykonywany i trwa tyle ile program jest wykonywany. Najbardziej popularnym jest wyświetlenie w przeglądarce danej strony.

Run time składa się z elementów JS Engine, WEB APIs(które są dostępne przez globalne okno) i call back queue(struktura danych zawierająca wszytkie call back queue gotowe do wykonania), jak evenhandler function.

A screenshot of a computer

Description automatically generated

Np. kiedy zostanie naciśniety przycisk, który zawiera event listener zostaje on wrzucony na listę z call back queue, w momencie, kiedy call stack jest pusty, zdarzenie z evenlistenera zostaje przesłane do niego do wykonania. Aby to mogło działać, w tle działa event loop, który wychwytuje nowe zdarzenia.

**Jak JS code jest wykonywany w Call Stack?**

A screenshot of a computer

Description automatically generated

Po przygotowaniu przez SJ engine gotowego kodu maszynowego, kolejnym krokiem jest tworzony **Global Execution Context for top level code**. **Execution context oznacza środowisko, w którym kod JS jest wykonywany i zawiera wszystkie potrzebne informacje do tego**. Top level code oznacza kod, który nie znajduje się w żadnej funkcji, czyli kod, który będzie wykonany jako pierwszy.

Jako przykład w prawej tabeli, top level code to stworzenie zmiennej „name”. Tworzone jest tylko jedno global execution context.

Następnie po stworzeniu global EC, top level code jest wykonywany.

Kolejno wykonywane są funkcje oraz oczekiwanie na wezwania (callbacks). Dla każdej funkcji zostaje stworzony execution context, zawierajacy potrzebne informacje do jej wykonania. Każdy execution context stanowi element call stacka.

**Co znajaduje się wewnątrz Execution Context?**

A screenshot of a computer program

Description automatically generated

Wewnątrz każdego execution context mamy:

* Variable enviorment, w którym mamy:
  + let, const and var declarations
  + Funkcje
  + Argumenty wprowadzone do obiektów
* Scope chain – składa się z odnośnik do zdeklarowanych let, const i var variables na zewnątrz
* this keyword – specjalne słowo pozwalające nam na wpływanie na stworzone obiekty. Jak self w pythonie w obiekcie.

Powyższe elementy są tworzone w fazie: creation phase, zaraz przed wykonaniem kodu.

Podsumowując, Call Stack to miejsce, gdzie execution contexts są zgromadzone w kolejności wykonywania ich.

A screenshot of a computer

Description automatically generated

Przez to że JS ma tylko jedną wiązkę egzekucyjną kodu, tworząc execution context global, dochodząc do const x, gdzie wzywamy funkcję fist (), następne do egzekucji jest funkcja first. Wewnątrz niej mamy const b, które stanowi wezwanie funkcji second i tutaj egezukcja execution context zostaje przesunięta z execution context first na execution context second.

**Scoping i Scope w JaveScript**

A screenshot of a computer

Description automatically generated

Scoping określa jak nasze zmienne są zorganizowane („Gdzie one się znajdują?”) i dostępne („Gdzie mamy do nich dostęp, a gdzie nie?”).

Lexical scoping określa jak scoping jest kontrolowany przez lokalizacje funkcji i bloków w kodzie.

Scope to miejsce albo środowisko, w którym konkretne zmienne są zadeklarowane (środowisko zmienne w przypadku funkcji). Mamy global scope, function scope oraz block scope.

A screenshot of a computer program

Description automatically generated

Scope Chain określa, który zakres ma do którego dostęp:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Call stack okresla w jakieś kolejności funkcje są wzywane, natomiast scope chain określa, który scope ma zakres do którego scopa.

**Hoist and TDZ (Temporal Dead Zone)**

Hoist – to sprawienie, że niektóre funkcje są dostepne przed ich deklaracją. Mechanizm ten polega na przeskanowaniu kodu przed jego wykonaniem i ustaleniu, które funkcje moga być wykonane.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

Temporal dead zone istenieje po to aby ułatwić nam unikanie/indetyfikacje błędów, które mogłyby się pojawić kiedy np. Stworzylibyśmy zmienną „job” a wykorzystali ją przed jej zdefiniowaniem.

**„This” keyword/variable**

A screenshot of a computer

Description automatically generated

This to specjalna zmienna tworzona dla każdego execution content. Nie jest statyczna, dlatego może zawierać różne parametry w zależności, gdzie zostaje użyta.

This keword in global scope to po prostu window:

Console.log(this) pokaże nam:

A screenshot of a computer screen

Description automatically generated

W prostej funkcji this ma wartość *undefined*.

A screenshot of a computer

Description automatically generated

W arrow function this ma wartość window dlatego, że pobiera wartość od swojego rodzica, w naszym przypadku global execution.

A screen shot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

Dla obiektów this keyword zawiera atrybuty obiektu:

A computer screen shot of a code

Description automatically generatedA screenshot of a computer

Description automatically generated

Ciekawa rzecz, możemy uzyć *method borrwing* pomiędzy dwoma obiektami. Gdybyśmy chwili wykorzystać to do przypisania metody do funkcji poza obiektem to by nie zadziałało, ponieważ reguła mówi, że *this* dla funciton declaration jest *undefined*.

A screen shot of a computer program

Description automatically generated

Gdybyśmy chcieli stworzyc w obiekcie methodę, która zawierałaby w sobie inną funkcję, to na to dwa rozwiązania.

Pierwsze przed stworzeniem wewnętrznej funkcji musimy stworzyć variable, któ®e zaweirałoby this dla danego obiektu.

Const self = this

function test () {

console.log(self.name)

}

Druga opcja to skorzystanie z arrow function, która nie posiada swojego własnego „this” tylko this swojego rodzica:

A screen shot of a computer program

Description automatically generated

**Primitive, Reference Objects in JaveScript Engine**

**A screen shot of a computer

Description automatically generated**

Tak jak się uczyłem, *primitive types* określa się liczby, string, boolean, null, undefined, symbol and bigint. Obiekty, również określa się jako *reference types* natomiast są: szyk, funkcje, obikety etc.

Primitive types są przechowywane w execution content w call stack, natomiast reference types są przechowywane heap Js engine.

A screenshot of a computer

Description automatically generated

**Jak wygląda zapisanie wartości variables w pamięci call stack a heap?**

Na poczatku jest tworzony indentyfikator, który wskazuje adress w pamięci w call stacku, pod którym przechowywana jest wartość.

W przypadku reference types, tworzony jest identyfikator, który wskazuje adress w call stacku, gdzie przechowywany jest adress z lokalizacja w pamięci heap danego obiektu.

**Dlaczego podczas stworzenia dwóch variables age i oldAge = age, a później zmianie age na inna wartość, po wydrukowaniu w konsoli mamy dwie różne wartości, a w przypadku takiej samy pracy na dwóch obiektach dostajemy taką sama „27”?**

Dzieje się tak ponieważ primitive types nalezą do grupy immutable, niezmiennych i w momencie tworzenia Age oraz oldAge mamy dwa identyfikatory wskazujące ten sam adress. Gdy zmieniliśmy wartość age na 31 to stworzyliśmy nowy adress zawierający wartość 31, wskazywaną przez indentyfikator age. Gdybyśmy zmieniliwartość oldAge, to stworzyłoby to nowy adress z nową wartością, a bucket usunąłby nieużwany adress w celu zwolnienia pamięci.

W przypadku reference value, wpływamy na zmiane wartości obiektu w heap a nie w call stack (pomimo wykorzystania const). Dzieje się tak ponieważ obydwa indentyfikatory wskazują na ten sam obiekt w heap.

Jeśli chcemy stworzyć nową pozycje w pamięci z refernce object możemy zastosować metodę z biblioteki Object.

A screen shot of a computer program

Description automatically generated

Jest tylko jeden problem z tą metodą, object assign sprawia, że dwa obiekty sa łączone w jeden na płytkim poziome. Gdybyśmy posiadali wewnatrz pierwszego obiektu inny obiekt np. Array, to ten array miałby jeden i ten sam odnosnik pomiędzy dwoma obiektami.

A computer screen shot of a code

Description automatically generatedA screen shot of a computer program

Description automatically generated

test