**PROGRAMOWANIE OBIEKTOWE (OOP)**

**Tworzenie obiektów i ich metod i własciwosci -** Do Tworzenia wielu można użyć 3:

1. **Factory Function** (tradycyjna metoda tworzenia od Mosha YT)

function zrobKolo (radius) {

return {

radius; //nie musze pisac radius=radius bo w nowym JS jeśli key=value to piszemy jedno.

rysuj: function() {

console.log(‘Rysuje koło!’);

**Nie uzywaj** tej metody. W kazdym obj. Funkcja rysuj będzie osobno kopiowana i ciazyla w pamieci.

} } }

const koło1= zrobKolo(5)

**B. Obiekt.create(**proto, [propObject]**)-**

***proto***- obiekt prototypu, z którego robimy nowy obj, ***propertiesObj*-**(opcj) Określa prop, które ma dodac do nowego obj.

const **Song** = {

Rzadko obecnie spotykana

    podsumowanie: function () {

        console.log(`${*this.*title} is written by ${*this.*author}.`);

    } }

const song1 = Object.create(**Song**); song1.title = "Hej Sokoły";

song1.author = "Marek Frankowski" //console.log(song1.podsumowanie())

| **Funkcja** | **Object.create()** | **new …()** |
| --- | --- | --- |
| Obiekt prototypowy | Jako prototyp przyjmuje istniejący obiekt | Dziedziczy z właściwości prototypu konstruktora |
| Funkcja konstruktora | Nie używany tutaj | Służy do tworzenia nowej instancji |
| Właściwości instancji | Nie ustawiono podczas tworzenia | Ustawiona za pomocą this wewnątrz konstruktora |
| Elastyczność | Bardziej elastyczny, umożliwia tworzenie obiektów o różnych prototypach | Mniej elastyczny, tworzy instancje określonej funkcji konstruktora |
| Składnia | Object.create(prototypeObject) | new ConstructorFunction(arguments) |
| Zwraca | Nowy obiekt(z okr.obiektem prototypowym i właściwościami) | Konstruktor obiektu lub obiekt |

1. **Przez konstruktor New+ fn()**

function **Book**(title, author, year) { //pierwsza litera nazwy z duzej!

*this.*title = title;

Wciąż stosowana. Częściej niż object.creator, ale nie tak jak classy

*this.*author = author;

*this.*year = year;

}

const book1 = new **Book**("Marymont", "Jonasz Kanovitch", 1999);

Słowo **new** powoduje automatyczne stworzenie nowego pustego obj., przypisuje do niego **this ,** aby na niego wskazywal i go zwraca.

Identyczny efekt( ale tylko wywolania) można uzyyskac przez uzycie **.call**- wywołuje funkcje z zadanym kontekstem (do którego kieruje This) i z zadanymi arg. **funkcja.call( kontekst, par1,par2..)** (Przykład jest w dalszejcz. tego dok).

1. **Class**

Tworzenie wzorca obiektu ( z właściwościami,wartościami, czynnościami) za pomocą **constructor**(){} oraz **this.** Wewnątrz constructora (poza defaultowym prototypem) definiujemy właściwości własne, kierujące this do tego obiektu, a poza construktorem są właściwości, które trafią do prototypu classy, które będą powszechnie dostępne dla każdego obieku tworzonego na bazie tej classy, z referencją do tego prototypu. Przez to nie kopiujemy ich dla każdego obiektu osobno.

Metody lepiej umieszczać w ciele klasy, a nie w constructorze. Wtedy będą one dostępne poprzez prototype. Identycznie jak w przypadku function constructor.

class Pokemon { //nazwa z duzej!

//jesli dla wszystkich obj. okreslamy od razu jakas wartosc to moge zapisac to nad konstruktorem, ale wtedy bez this

    health = 100;

// **constructor** słuzy do inicjalizacji tj. tworzenia instancji klasy, czyli do tworzenia obiektu na podstawie tej klasy. Jego zapis jest jak przy funkcji bo to metoda

**constructor(name, type, coach)** {

*this.*name = name;

*this.*type = type;

*this.*coach = coach;

    }

    attack() { //mogę tez definiowac czynnosci ktore wykona(metody) ale lepiej poza constructorem…

        console.log('Haaaadziaa!');//… bo wtedy będą dostępne również dla innych obiektow tworzonych z tej classy

*this.*health -= 20;

    }

    doubleAttack() { //odwolanie sie do jakiejs metody w obj.

*this.*attack()

*this.*attack()

    }

}

const pikachu = new Pokemon("Pikachu", "electric") // tworze nowy obiekt

console.log(pikachu.hasOwnProperty(‘friends’) // false. Ale właściwość hasOwnProperty istnieje,ale nie jest wlasna tylko z prototypu

**POD-CLASSY (SUBCLASS) -** Jest to funkcja w OOP, w której klasa dziedziczy funkcje z klasy nadrzędnej, ale posiada dodatkowe funkcje, których nie ma rodzic, np. mamy główną class’e Zwierzęta(name, age, species) i chce utworzyć class Koty. Zamiast tworzyć ją od zera - podając na nowo name , age i species , odziedziczę je po class’ie Zwierzęta . Klasa kotów może wtedy mieć dodatkowe wlasne właściwości( np. kolor wąsów) .

**UZYCIE CLASS do modyfikacji DOM i HTML (freeCodeCamp)**

//w tej klasie bedziemy mogli przekazywac moja liste do elementu konstruktora. Classa ta zachowuje referencje do znacznika html z moja lista(cala lista jako element)

**class** modyfikacjaListy {

constructor(element) {

**this**.listaElementow = element;//listElem-odnosi sie do listy html-calej <ul>,nie tylko <li>

**this**.textList = []//ta arr bedzie zawierac stringi ktore wstawie do list jaki pojedyncze <li>

}

**static** stworzElementListy(text) {

**const** li = document.createElement('li')//w () wpisuje typ selectora,bo crea..sluzy do ich tworzenia li.textContent = text;//po stworzeniu znacznikow wstawiamy w zawartosc text z arg

**return** li;

}

aktualizuj() { //ustawiamy metode w tej klasie ktora bedzie odswiezac zawartosc listy

**while** (**this**.listaElementow.firstChild) {//dopki true to działaj,tj. dopóki(sa El. Na liscie) **this**.listaElementow.removeChild(**this**.listaElementow.firstChild)//usun i tak az nie brakni

}

**for** (**const** text of **this**.textList) { //wstawiamy li jesli jakies sa w tablicy textList

**this**.listaElementow.appendChild(modyfikacjaListy.stworzElementListy(text))

}

}

dodaj(text) { //ustawiamy metode, ktore dodaje do listy wpisany tekst

**this**.textList.push(text); //dodajemy to tablicy text

**this**.aktualizuj(); //od razu po uzyciu metody dodaj uruchamiam aktualizacje listy

}

usun(index) { //ustawiamy metode, ktora usuwa z listy pozycje z wskazanego indexu

**this**.textList.splice(index, 1); // (usuwamy od wskazanego indexu, ile indexow (1))

**this**.aktualizuj();

}

}

**const** myList = document.getElementById('myList'); //querujemy moja cala liste w html

**const** nowaListaModyfikacja = **new** modyfikacjaListy(myList);//nowaL…Modyfikacja mojej listy HTML z classa ..

// nowaListaModyfikacja.aktualizuj() … listBinding

// nowaListaModyfikacja.dodaj('przykladowy tekst')

// nowaListaModyfikacja.usun(2) //usuwa index 2

**Lista koncepcji najczęściej używana, gdy mówimy o programowaniu obiektowym:**

* 1. **Hermetyzacja** (Encapsulation)
* 2. **Abstrakcja**
* 3**.** Ponowne użyce i **dziedziczenie** (Reusability/inheritance)
* 4.Wielopostaciowość **(Polymorphism)**
* 5. Stowarzyszenie/ Skojarzenie (Association)
* 6. Zbiór (Aggregation)
* 7. Kompozycja

**Ad.1 Enkapsulacja (Hermetyzacja)** – gromadzenie danych i metod w obrębie jednostki. Zapewnia zmnieszenie zapisu, zwiększenie czytelności kodu i możliwość wielokrotnego jego użycia. Polega ono na rozdzieleniu wewnętrznego i zewnętrznego interfejsu naszego obiektu. Tu właśnie pojawia się pojęcie hermetyzacji, która polega na odpowiednim ukrywaniu pewnych składowych przed zewnętrznym środowiskiem. Nasz obiekt może z nich korzystać, natomiast zewnętrzne środowisko może używać tylko rzeczy, które my mu świadomie udostępnimy.W Abstrakcji znaliśmy co robi, ale nie w jaki sposób. W hermetyzacji natomiast posiadamy już wiedzę o implementacji, ale chcemy ją ukryć przed światem zewnętrznym posługując się własnie systemem, modułem, klasą czy metodą. Stanowi pewnego rodzaju barierę. Zazwyczaj możemy to osiągnąć poprzez stosowanie modyfikatorów dostępu, takich jak public, private i protected, które określają, które pola i metody są widoczne dla innych obiektów i które są ukryte. Użytkownik ma do nich dostęp tylko i wyłącznie za pomocą ściśle określonych mechanizmów. Nie jest w stanie ich bezpośrednio odczytywać, zmieniać lub modyfikować. Hermetyzacja ukrywa również logikę, której użytkownik końcowy nie musi lub nie powinien być świadom.Nie jest upoważniony do bezpośredniego korzystania ze wszystkich modułów znajdujących się w naszym serwerze, dzięki czemu nie może niczego popsuć, ani ominąć zabezpieczeń oraz reguł biznesowych. Następnie do class w tych modułach, potem metod w tych classach itd.

**Ad.2 Abstrakcja-** Abstrakcja upraszcza nam pewne rzeczy przez usunięcie informacji, które nie są niezbędne do ich zrozumienia. Wiemy co robi, ale nie w jaki sposób.Przykładowo aplikacja z której korzysta użytkownik jest abstrakcją, która ukrywa różne moduły, które z kolei mogą ukrywać klasy, które z kolei składają się z metod. Nawet sam język programowania jest warstwą abstrakcji dla systemu binarnego, które z kolei jest warstwą abstrakcji dla fluktuacji strukturalnych w polu magnetycznym itd, itp. Warstw abstrakcji jest nieskończenie wiele. Jak możesz zauważyć im wyższa warstwa abstrakcji tym mniejsza złożoność naszego systemu. Użytkownik aplikacji nie musi się przejmować w jaki sposób oraz w jakiej kolejności są wywoływane niskopoziomowe metody.

Z abstrakcja w OOP wiążą się pojecia **getter** i **setter** omówione na kolejnych stronach.

const Game = function (getTitle, getAuthor) {

    let title = getTitle; //PRIVATE zmienne title i author

    let author = getAuthor;

    this.giveTitle = function () { //PUBLIC method z dostepem do privat title

        return title;

    }

    this.giveAuthor = function () {  //PUBLIC metod z dostepem do privat autor

        return author;

    }

    const pigulka = function () { //PRIVATE method to get pigulka

        return `${title} zostało stworzone przez ${author}`;

    }

    this.givePigulka = function () { //PUBLIC metod ktora ma dostep do privat metody.Uwaga z funkcjami na nawiasy ()!        return pigulka();

    }

}

const game1 = new Game('Wiedzmin', 'CD Project')

console.log(game1.giveTitle()) //wiedzmin

console.log(game1.giveAuthor()) //CD Project

//console.log(game1.pigulka())// TypeError: ..pigulka is not a fn

console.log(game1.givePigulka())

### Abstrakcja vs Hermetyzacja (https://drycode.pl/blog/cztery-cechy-oop/)

Często ludzie, mylą abstrakcje z hermetyzacją, ponieważ ich definicje brzmią bardzo podobnie. Zapytani czym jest abstrakcja odpowiadają, że jest to jedna z czterech cech OOP, która ukrywa nieistotne dla nas informacje i skupia się tylko na tym czego potrzebujemy. Zapytani czym jest zatem hermetyzacja odpowiadają, że jest to kolejna cecha, która również ukrywa detale. kluczowe jest to na jakim poziomie to robią. Abstrakcja ukrywa detale na poziomie designu (nie musimy tutaj wiedzieć nic o szczegółach implementacyjnych). Możemy zdefiniować abstrakcje nie mając o nich zupełnie pojęcia. Natomiast hermetyzacja ukrywa informacje na temat już zaimplementowanych szczegółów. Prościej mówiąc hermetyzacja ukrywa faktyczną implementacje.

Spójrzmy na wcześniej wspomniany ekspres. Abstrakcją jest tutaj fakt, że patrząc na ekspres widzimy ekspres, a nie części z których się składa. Widzimy uproszczony obraz ekspresu, który rozumiemy. Jeśli ktoś wymieni części znajdujące się w środku my nie zauważymy zupełnie żadnej różnicy. Hermetyzacja natomiast to ukrycie tych wszystkich części w środku. Dostęp do nich jest możliwy jedynie przez udostępniony interfejs w postaci ekranu lub przycisków. Dzięki temu nie mamy bezpośredniej możliwości oddziaływania na poszczególne części, przez co stają się one odporne na czynniki zewnętrzne.

|  |  |
| --- | --- |
| **Abstrakcja** rozwiązuje problem na poziomie designu | **Hermetyzacja** rozwiązuje problemy na poziomie implementacji |
| Abstrakcja służy do ukrywania niepotrzebnych informacji i skupia się na istotnych kwestiach | Hermetyzacja ukrywa kod i dane w jednej jednostce przez co chroni je przed światem zewnętrznym |
| Abstrakcja pozwala skupić się na tym co obiekt robi zamiast tego jak to robi | Hermetyzacja ukrywa detale i mechanizmy, które są odpowiedzialne za to jak obiekt coś robi |
| Abstrakcja jest bardziej ogólnym spojrzeniem na obiekt podczas desingu.  Np. patrzac na telefon to widzimy, że ma ekran oraz przyciski, które służą do wybierania numeru. | Hermetyzacja ukrywa szczegóły impementacyjne odpowiedzialne za działanie  Np. ukrywa wewnętrzne szczegóły implementacyjne telefonu: w jaki sposób działa klawia i ekran,jak są ze sobą połączone |

**Ad.3 Możliwość ponownego użycia/dziedziczenie -** Dziedziczenie JS to mechanizm, który pozwala stworzyć nową klasę przy użyciu istniejącej klasy. Oznacza to, że klasa potomna dziedziczy wszystkie właściwości i zachowania klasy nadrzędnej. Dziedziczenie jest mechanizmem, w którym jedna klasa nabywa własności (właściwości i metody) innej klasy po której dziedziczy. Dzięki dziedziczeniu możemy ponownie wykorzystać pola i metody istniejącej klasy bez konieczności ich ponownej implementacji. Klasa po której dziedziczymy często jest nazywana super klasą, lub klasą rodzicem. Natomiast klasa, która dziedziczy po super klasie często jest nazywana subklasą lub klasą dzieckiem.

Zasadniczo JS nie jest językiem opartym na klasach. *Słowo CLASS* zostało wprowadzone w ES6, ale jest cukrem składniowym, JavaScript pozostaje oparty na prototypach. W JavaScript dziedziczenie odbywa się za pomocą prototypu. Ten wzorzec nazywa się ***wzorcem delegowania zachowań*** lub **dziedziczeniem prototypowym**.

**class** Person {//główna classa-rodzic

constructor(name, age) {

**this**.name = name;

**this**.age = age;

}

opisz() {

**return** (`Jestem ${**this**.name} i mam ${**this**.age} lat.`)

}

}

**class** Programmer **extends** Person {//extends - zapis dziedziczenia, classa potomek, rozszerza classe rodzica,

constructor(name, age, lataDoswiadczenia) { // tzn,przyjmuje własc. z Person i do tego ma swoje.

**super**(name, age) //wywolujemy arg. z rodzica

**this**.lataDoswiadczenia = lataDoswiadczenia; //a tu tworze jego wlasne wlasciwosci

}

code() {

console.**log**(`${**this**.name} właśnie koduje.`)

}

}

**const** person1 = **new** Person('Marek', 44);

**const** programmer1 = **new** Programmer('Jan', 37, 9)

console.**log**(programmer1.opisz()) // Dziala , bo progr.. korzysta z wlasciwosci z persona, natomiast..

console.log(person1.code())//.. to nie dziala bo code jest indywid. zachowaniem programmera

inny przykład…

const Artist = function (name2, country) {

    this.name2 = name2; //2 bo na poczatku uzylem juz taka zmienna

    this.country = country;

    this.greeting = () => console.log('elo!');

}

const VanGogh = new Artist('Vincet', 'Nederland')

console.log(VanGogh.greeting())

//powyzsza instancja dla klasy Artist czyli VanGogh ma te same wlasciwosci i metode co classa, wiec to jest dziedziczenie, i widac ze dziala.

/\*Powiedzmy sobie teraz nieco więcej o prototypach.W dwóch słowach, prototypy są to wlasciwosci bądź metody, które są odziedziczone od konstruktora, jednak nie są bezpośrednio dostępne jako property(bądź metoda) instancji.\*/

Artist.prototype.login = () => console.log("Jestem zalogowany!");

**O Prototypie**

**1 z 2**

Console.log(VanGogh) // -->

Artist { name2: 'Vincet', country: 'Nederland', greeting: ƒ }

country:"Nederland"

>greeting:() => console.log('elo!')  //metoda nalezaca do instancji VanG.

name2:"Vincet"

>[[Prototype]]:Object // wlasc.i nalezace do prototypu,ale niedostepne bezposrednio jako wł.instancji

**Ad.4 Wielopostaciowość / Polimorfizm –** zdolność do wywoływania tej samej metody na roznych obiektach (wewnątrz pochodnej classy potomnej rodzica) i reagowania na swój wlasny sposób. Zdolnosc do wywollania tej samej metody co u classy-rodzica, ale przez classe potomna , można również ja wywoływać z inna wartoscia(zmiana).

**PRZYKŁAD 1** – uzice metody rodzica z wartoscia potomka

**class** Animal {

constructor(name) {

**this**.name = name;

}

makeSound() {

console.**log**(`${**this**.name} robi ryk ogolny!`);

}

}

**const** animal1 = **new** Animal('Lew') console.**log**(animal1.makeSound())

**class** Dog **extends** Animal {

constructor(name) {

**super**(name);

}

makeSound() {

console.**log**(`Hau hau hau!!!`);

} //classa potomka uzywa metody makeSound rodzica ale na swoj sposob(inny dzwiek). Czyli

} podmienia dzwiek rodzica na swoj. Gdyby nie mial swojego sposobu to by mial sposob rodzica- tj. **łańcuch**

**const** dog1 = **new** Dog('Burek'); console.**log**(dog1.makeSound()) … **prototypowy**

**PRZYKŁAD 2** – uzycie metody rodzica, ale uzycie jednoczesnie wartosci rodzica i potomka.

**class** Warzywo {

constructor(typ) {

**this**.typ = typ;

}

rosnij() {

onsole.**log**('rosne rosne!')

}

}

**class** Burak **extends** Warzywo {

constructor(typ) {

**super**(typ)

}

rosnij() {

**super**.rosnij();//uzycie tego od rodzica

console.**log**('Ale tak wolniutko...')//i cos swego

} } **const** burak1 = **new** Burak('buraczek'); console.**log**(burak1.rosnij())

**Ad.5 Asocjacja/Skojarzenie-** Jest rodzajem relacji między obiektami., w której obiekty są od siebie niezależne. Oznacza to, że usunięcie związku pomiędzy obiektami **nie wpływa na ich sposób funkcjonowania**.

function Person(name) { //fn tworzaca osobe

    this.name = name;

    this.phone = null;

}

function Phone(phone) { //fn tworzaca tel

    this.phone = phone;

}

const john = new Person('John');    //tworzymy johna

const phone = new Phone('iPhone');  //tworzymy obiekt Phone

john.phone = phone;//nadajmy zwiazek John.phone z Phonem.

console.log(john.phone) // iphone

john.phone = 'motorolla'; //usuwam związek

console.log(john.phone) // motorolla

**Ad.6 Agregacja- Agregacja jest szczególnym przypadkiem asocjacji**. Dotyczy ona sytuacji, w której jeden obiekt pełni funkcję nadrzędną względem drugiego. Obiekt właściciela jest często nazywany **agregatem**, a obiekt będący własnością nazywany jest **komponentem**. Agregacja jest również nazywana relacją **„Has-a”**. W poniższym przykładzie obiekty typu Person pomagają w budowaniu obiektu university. W przypadku zamknięcia uniwersytetu, studenci nie przestaną istnieć. Innymi słowy agregacja polega na tworzeniu obiektu składającego się z innych obiektów, które są związane z obiektem nadrzędnym, jednak mimo to mogą istnieć niezależnie.

Const university= {

Students: []

};

Const marek = new Person('marek'); // na bazie fn Person z pkt7

university.students.push(marek);

**Ad.7 Kompozycja- Kompozycja jest z kolei szczególnym przypadkiem agregacji**. W tym przypadku żaden obiekt składowy nie może istnieć poza swoim obiektem nadrzędnym.

const John = {

firstName: "John",

address: {

street: "Piłsudskiego 23",

city: "Wrocław"

}

};

Obiekt reprezentujący adres w tej sytuacji jest ściśle związany z obiektem john i jego istnienie w przypadku skasowania obiektu nadrzędnego nie ma sensu.

Związane OOP, ale dodalem od siebie

**PROTOTYP –** Właściwość  \_\_proto\_\_\_ (old) lub <prototype>: Object{ … }

Wskazuje na obj., który jest używany jako prototyp. Każdy obiekt defaul.ma tę właściwość, która odnosi się do **Object Protoype**. Wyjątkiem, jest syt, gdy ustawiono inaczej (to znaczy, gdy obiekt \_\_proto\_\_ wskazuje na inny prototyp). Np.

function DogObject(name, age) {

let dog = Object.**create**(constructorObject); //Przez metodę Object.create() ustawiamy dla **obiektu**

dog.name = name; // … **dog** jako **prototyp** obiekt **constructorObject.**

dog.age = age;

return dog;

}

**O Prototypie**

**2 z 2**

let constructorObject = {

speak: function(){

return "I am a dog"

}

}

let bingo = DogObject("Bingo", 54);

console.log(bingo);

console.log(bingo.name); //Bingo

console.log(bingo.speak()); // I am a dog

Object.**get**PrototypeOf(obj) – statyczna metoda, **zwraca** prototyp [[prototype]] danego obiektu (czyli ‘przypisaną’ właściwość) lub null. Uwaga! W ES5 zgłosi [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError)wyjątek, jeśli obj parametr nie jest obiektem. W ES2015 parametr zostanie przekształcony w plik [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object).

const proto = {};

const obj = Object.create(proto); //obiekt tworzony na podstawie innego istniejącego obiektu o nazwie proto

Object.getPrototypeOf(obj); // proto

Object.getPrototypeOf(obj) === proto; // mogę sprawdzic, czy prototypem obiektu **obj.**, jest obiekt **proto**

Object.**set**PrototypeOf(obj, prototype) – statyczna metoda, **ustawia** prototyp [[prototype]] danego obiektu (czyli ‘przypisaną’ właściwość) na inny obiekt, lub null. Obj= obiekt, któremu ustawiamy prototyp= nowy prototyp tego obiektu. Zwraca obiekt, któremu ustawiamy prototyp.

Jeśli obj parametr nie jest obiektem (np. liczbą, ciągiem znaków itp.), ta metoda nic nie robi (może wyskoczyc blad) i zwraca bezpośrednio jako wartość pierwotną obj. Jeśli prototype ma taką samą wartość jak prototyp obj, to obj jest zwracany bezpośrednio, bez powodowania TypeError nawet wtedy, gdy obj ma niezmienny prototyp.  
Uwaga! [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError) jest wyrzucane, jeśli spełniony jest jeden z poniższych warunków:

-Parametrem obj jest undefined lub null.

-Parametr objjest [nierozszerzalny](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible) lub jest [niezmiennym prototypowym obiektem egzotycznym](https://tc39.es/ecma262/#sec-immutable-prototype-exotic-objects), (Object.prototype lub [window](https://developer.mozilla.org/en-US/docs/Web/API/Window)). Jednak błąd nie jest zgłaszany, jeśli nowy prototyp ma taką samą wartość jak oryginalny prototyp pliku obj.

-Parametr prototype nie jest obiektem ani plikiem [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null).

function.call()- wywołuje funkcje z zadanym kontekstem(do którego kieruje This) oraz z zadanymi argumentami. Jej uzycie ma sens np. gdy chcemy uzyc jakiejs funkcji, ale do jakiegos konkretnego, np. zmienionego kontekstu lub dla innych argumentow. Argumenty podajemy pojedynczo. **funkcja.call( kontekst, arg1, arg2 ...)**

**const** pracownik = {

zawolaj: **function** (name) { // uzywam fn **zawolaj** z Pracownika, dla Person1 i 2

console.**log**(`Ahoj! Jestem ${name}.`) // Obiekt.function.call(kontekst, argument)

}

}

**const** Person1 = { **const** Person2 = {

name: 'Janusz', name: "Artur",

age: 20, age: 37,

} }

**const** wynik1 = Pracownik.zawolaj.**call**(Person1, Person1.name) // ”Ahoj! Jestem Janusz.”

**const** wynik2 = Pracownik.zawolaj.**call**(Person2, Person2.name) // ” Ahoj! Jestem Artur.”

<http://blog.nebula.us/12-javascript-wykorzystanie-metod-call-oraz-apply-w-praktyce>

function.apply()- robi dokladnie co .call, ale argumenty podajemy w tabeli. **funkcja.apply( kontekst, [1,2,3] )**

**static metod-**metoda , uznawana za jakby funkcję pomocniczą, którą można zdefiniować w classie. Nie potrzebuje instancji, bo możey ją wywołać bezpośrednio na klasie. (Patrz tez przykład z modyf. DOM i HTML)

class **NazwaClassy** {

constructor( arg.) {

jakies this. = ….;

}

**Static** funkcyjaN(arg) {

return ..cos robie

}

}

Const nowyObj1= new NazwaClassy(arg);

**NazwaClassy**.funkcyjaN(arg) // na classie a nie instancji (nowyObj1)

**Gettery i settery** są metodami (tzw. Akcesoria) uzywane do pobrania/zmiany wartości prywatnego pola klasy z zewnątrz i użycia/wywołania ich jak właściwość obiektu. Getter zwraca wartość konkr.pola, a setter ustawia wartość.(opis w pliku Mosha)

 function Kwadrat5(bok) {

    let color = "niebieski";

    this.bok = bok;

    this.**getColor** = function () { //getter!

        return color;

    }

Object.defineProperty(this, 'color', {

**get:** function () {

            return color; // do wywolania jako color

        },

**set:** function (value) { //jeśli zrobimy Obiekt.color = „inny niż niebieski”to będzie error!:)

            if (value !== 'niebieski')

                throw new Error('Zly kolor');

        }   });

Object.freeze(obiektDoNiezmiennialnosci) – metoda zapobiega modyfikowaniu istniejących właściwości, oraz przez dodawaniem nowych. (wyjaśnione tez na wcześniejszych kartkach z immutability itd.)

**Strict mode-** Strict mode to funkcjonalność wprowadzona w ES5 (ECMAScript 2009), która pozwala odpalać nasz skrypt w bardziej restrykcyjnym trybie - tak zwanym "strict mode". Gdy nasz kod piszemy z wykorzystaniem [modułów w ES6](https://kursjs.pl/kurs/es6/moduly) lub wykorzystując [klasy](https://kursjs.pl/kurs/obiekty/class), nie musimy wtedy używać powyższego przełącznika, ponieważ w obydwu przypadkach strict mode jest włączone. Tryb ten eliminuje niektóre "ciche" błędy (takie, które nie są sygnalizowane przez przeglądarkę), lepiej sygnalizuje powszechne kodowe wpadki oraz niebezpieczne operacje. Domyślnie tryb ten jest wyłączony. Aby go włączyć, na początku naszego kodu musimy użyć zapisu "**use strict**".

Różnice między trybem ścisłym, a klasycznym:

- brak możliwości używania zmiennych bez ich zadeklarowania (test=20 console.log(test)// 20)

- pomijanie słów kluczowym przy pętlach (for( **let** i=0,…)

-sygnalizowanie błędu przy próbie nadpisania właściwości obj. Oznaczonych jako nie do zapisu

-Usuwanie funkcji, zmiennych czy argumentów nie jest już możliwe. Możemy usuwać tylko właściwości obiektów, ale tylko te, które są bezpieczne do usunięcia.

-zabronione używanie słów kluczowych dla nazw zmiennych (var **let=20**) //let = 20)

-parametry funkcji nie mogą się powtarzać

-Deklarując liczby, nie możemy poprzedzać je cyfrą 0

-funkcji `eval()`, która służy do uruchamiania kodu, który podajemy jako jej parametr w formie stringu. W domyślnym trybie powyższy kod stworzył by zmienną x dla otaczającego go zasięgu. W przypadku użycia "strict mode", kod wewnątrz eval tworzy swój własny zasięg dla jego zmiennych.

Taki tryb możemy włączać nie tylko dla całego kodu, ale i dla poszczególnych funkcji umieszczając go na ich początku:

// kod bez strict-mode...

(function(){

"use strict";

// kod działający na strict-mode

})(); // kod bez strict-mode...