Object Oriented Programming

Autor:

Koncepcje związane z OOP

Autor:

Definicja OOP

Programowanie obiektowe (OOP) jest paradygmatem programowania, w którym korzystamy z obiektów, które mają własne właściwości, takie jak:

- pola (dane, informacje o stronie)
- metody (działania / funkcje, które wykonuje obiekt)

W programowaniu obiektowym definiujemy obiekty i wywołując ich metody, umożliwiamy ich wspólną interakcję.

Koncepcje OOP - abstrakcja

Pojęcie abstrakcji w programowaniu obiektowym jest najczęściej utożsamiane z klasami (w ES5 nie ma klas).

Klasa to model, który tak naprawdę nie reprezentuje żadnego istniejącego obiektu. Klasa jest tylko podstawą (szablonem) do definiowania i tworzenia nowych obiektów.

Przykładem jest samochód lub ryba - nie są to obiekty z prawdziwego świata. Obiektami świata rzeczywistego na podstawie ich właściwości są np .: Fiat 126p lub szczupak.

Koncepcje OOP - dziedziczenie

Technika programowania, która pozwala na wykorzystanie istniejącej klasy do utworzenia nowych, na podstawie elementu nadrzędnego.

W JS, gdzie nie ma pojęcia samej klasy, obiekty dziedziczą po innych obiektach, i nazywa się to dziedziczeniem prototypowym.

Koncepcje OOP – polimorfizm

Podczas pisania programu wygodnie jest traktować nawet różne dane w jednolity sposób.

Niezależnie od tego, czy powinieneś dodać liczbę czy tablicę, zwykle jest bardziej czytelne, gdy operacja nazywa się po prostu add, a nie addNumber lub addString.

Jednak ciąg tekstowy musi być dodany inaczej niż tablica (jest to konkatenacja, a nie dodawanie), więc będą dwie implementacje dodawania, ale nazywanie ich wspólną nazwą tworzy wygodny interfejs niezależny od podawanego typu wartości.

Koncepcje OOP – enkapsulacja / hermetyzacja

Hermetyzacja zapewnia, że obiekt A nie może nieoczekiwanie zmienić stanu obiektu B

Tylko metody obiektu B są uprawnione do zmiany jego własnego stanu.

Każdy obiekt udostępnia swój publiczny zestaw metod, z których inne obiekty mogą korzystać do zmiany stanu tego obiektu.

Autor:

Koncepcje OOP – zalety i wady

Zalety:

+ łatwy do zrozumienia
+ łatwa modyfikacja kodu
+ możliwość pracy wielu programistów na tym samym kodzie
+ łatwa rozbudowa, elastyczność
+ możliwość ponownego wykorzystania raz napisanego kodu

Niedogodności:

- bardziej abstrakcyjne podejście, w którym potrzebne jest więcej myślenia
- w niektórych przypadkach jest to niepotrzebne

Kontekst wywołania funkcji

Autor:

Kontekst wywołania funkcji – słowo kluczowe ‚this’

W większości przypadków wartość **this** zależy od sposobu wywołania funkcji i wskazuje kontekst wykonania funkcji. Jeśli napiszemy normalną funkcję, będzie to działało w ten sposób (funkcje strzałek ES6 zachowują się inaczej!).

W kontekście globalnego wykonania odnosi się to do obiektu globalnego - w przeglądarkach - okna.

Nie można ustawiać kontekstu podczas wykonywania funkcji (można to zrobić zanim wywołamy funkcję)!

Kontekst (wartość **this**) może być inny przy każdym wywołaniu funkcji!

Możemy ustawić kontekst wykonania za pomocą metod .bind (), .call (), apply ().

Kontekst wywołania funkcji – słowo kluczowe ‚this’

```
function sayHello() {
        console.log(this.name + " mówi hej!");
}

const person = {
        name: "Zenek",
         sayHello : sayHello
};

person. sayHello() // Zenek mówi hej!

sayHello() // Undefined mówi hej!
```

Autor:

ZADANIE 1

1. Spróbuj za pomocą console.log wyświetlić wartość ***this***. Zrób to w zakresie globalnym oraz w zakresie funkcji(napisz dowoloną funkcję i wywołaj ją)

2. Dołącz tą funkcję do nowo stworzonego obiektu. Nazwij ją jako metodę tego obiektu i wywołaj.

Kontekst wywołania funkcji – bind()

Tworzy kopię funkcji, która ma tę samą zawartość oraz parametry co funkcja oryginalna ale otrzymuje nowy kontekst podany jako parametr, który zostaje na stałe powiązany z tą funkcją.

```
function returnX () {
        return this.x;
}

const obj = { x: 42 }

returnX () // wywołanie funkcji w zakresie globalnym, zwracana jest wartość „undefined”

const boundReturnX = returnX.bind (obj)

boundReturnX () // wywołanie funkcji w kontekście obiektu „obj” zwróci 42
```

ZADANIE 2

Stwórz funkcję która za pomocą console.log() wyświetla wartość ***this***.

Stwórz obiekt.

Powiąż funkcję z obiektem za pomocą metody bind().

Wywołaj oryginalną funkcje w zakresie globalnym oraz nową kopię funkcji w zakresie obiektu, sprawdź co zwróci console.log()

Kontekst wywołania funkcji – call()

call() wywołuje funkcję z kontekstem podanym jako pierwszy argument metody **call()** oraz argumentami dla danej funkcji wymienionymi jako następne argumenty, oddzielone przecinkiem.

```
function returnSum (a, b) {
        return this.x + a + b;
}

const obj = {x: 42};

returnSum.call (this, 1, 1); // NaN
returnSum.call (obj, 1, 1); // 44

returnSum.call ({}, 1, 1); // NaN
returnSum.call ({x: 142}, 1, 1); // 144
```

Kontekst wywołania funkcji – apply()

apply() działa podobnie do **call()**. Wywołuje funkcję z kontekstem podanym jako pierwszy argument, natomiast argumenty dla samej funkcji przekazujemy w postaci tablicy.

```
funkcja returnSum (a, b) {
        return this.x + a + b;
}

const obj = {x: 42};

returnSum.apply (this, [1, 1]) // NaN
returnSum.apply (obj, [1, 1]) // 44

returnSum.apply ({}, [1, 1]) // NaN
returnSum.apply ({x: 142}, [1, 1]) // 144
```

ZADANIE 3

Napisz funkcję, która za pomocą pierwszego **console.log()** pokazuje wszystkie argumenty funkcji, a w drugim **console.log()** wartość **this**

Wywołaj ją za pomocą **call()** i **apply(),** przekazując obiekt jako kontekst i przykładowe argumenty do wyświetlenia.

Tworzenie obiektów i dziedziczenie w JavaScript

Autor:

Tworzenie obiektów i dziedziczenie – literał obiektu

Najprostrzym sposobem na utworzenie obiektu jest utworzenie literału obiektu:

```
const cat = {
        name : “Fluffy”,
        age: 1,
        sound: “Meeeeow!”,

        makeSound: function(){
                console.log(this.sound)
        },
        speak: function(){
                console.log(‘Sorry cats can’t speak’)
        }
};
```

Tworzenie obiektów i dziedziczenie – funkcja fabryki

Funkcja fabryki to funkcja, która tworzy coś za pomocą argumentów przekazywanych do funkcji. Zwykle tworzy obiekt:

```
function catFactory(name, age) {
   return {
      name : name,
      age: age,
      sound: “Meeeeow!”,
      makeSound: function(){ console.log(this.sound) },
      speak: function(){
         console.log(‘Sorry cats can’t speak’)
      }
   }
}
const cat1 = catFactory(’fluffy’, 2);
const cat2 = catFactory(’Garfield’, 3);
```

ZADANIE 4

Utwórz funkcję fabryki, która produkuje obiekt cat z właściwością **name** podawaną jako argument funkcji.

Utwórz kilka kotów i przypisz je do zmiennych.

ZADANIE 5

Rozszerz funkcję z zadania 4.

Teraz również chcemy przekazywać właściwość **sound** obiektu **cat** jako argument, chcemy mieć metodę „**makeSound**”, która rejestruje **this.sound**

Tworzenie obiektów i dziedziczenie – czym jest referencja?

W JavaScript tworząc nową zmienną i ustawiając jej wartość jako istniejącą funkcję, obiekt lub tablicę - nie tworzymy nowej kopii, lecz przekazujemy referencję do istniejącego zasobu.

```
const obj1 = { nazwa: 'Fioletowy wieloryb’ }
const obj2 = obj1; // nie tworzymy nowego obiektu, przekazujemy tylko referencję do obiektu obj1

console.log(obj1 === obj2) // true

obj2.nazwa = 'Fioletowy jeleń’;

console.log(obj1); // { nazwa: 'Fioletowy jeleń’
```

Zmienna obj2 wskazuję na ten sam obiekt co zmienna obj1.s

Tworzenie obiektów i dziedziczenie – czym są prototypy?

Jeśli chodzi o dziedziczenie, JavaScript ma tylko jedną konstrukcję: obiekty. Każdy obiekt ma specjalną własność prywatną, która zawiera referencję do innego obiektu zwanego jego prototypem. Ten prototypowy obiekt ma własny prototyp i tak dalej, w górę łańcucha dziedziczenia dopóki nie zostanie osiągnięty *null* dla obiektu znajdującego się na samej górze łańcucha dziedziczenia. Z definicji *null* nie posiada prototypu i działa jako ostatnie ogniwo w tym łańcuchu prototypów.

Większość obiektów w JavaScript są instancjami dziedziczącymi po obiekcie nadrzędnym: Object, który znajduje się na szczycie łańcucha prototypów.

Ta cecha jest często uważana za jedną ze słabości JavaScript ponieważ jest inna niż w innych językach obiektowych co wymaga delikatnej zmiany myślenia . W rzeczywistości prototypowy model dziedziczenia jest dużo bardziej elastyczny niż model klasyczny. Na przykład zbudowanie klasycznego modelu na podstawie modelu prototypowego jest dość trywialne.

Tworzenie obiektów i dziedziczenie – __proto__ & prototype

__proto__ to wewnętrzna właściwość każdego nowego obiektu, której silnik JS używa do przechowywania referencji do prototypu obiektu konstruktora. Nigdy nie powinieneś mieć do niego bezpośredniego dostępu! Według specyfikacji języka **__proto__** nowo utworzonego obiektu ustawia referencję na **prototype** funkcji konstruktora która stworzyła ten obiekt.

prototype jest „normalną” właściwością, którą mają wszystkie funkcje. Gdy funkcja jest używana jako konstruktor ze słowem kluczowym **„new”**, obiekt przechowywany we właściwości **prototype** jest ustawiany jako prototyp nowo utworzonego obiektu (**__proto__**).

NIE POMYLCIE SIĘ! ;)

Autor:

Tworzenie obiektów i dziedziczenie – słowo kluczowe „new”

Funkcja konstruktora to funkcja napisana w celu wywołania ze słowem kluczowym ***new***.

Słowo kluczowe „***new***” robi 4 rzeczy z funkcją:

1. Tworzy nowy pusty obiekt
2. Ustawia prototyp (pole **__proto__**) tego obiektu, aby wskazywał na **prototype** funkcji konstruktora
3. Funkcja konstruktora wywołuje się w kontekście nowo utworzonego obiektu
4. Zwraca obiekt utworzony w kroku 1.

Tworzenie obiektów i dziedziczenie – funkcja konstruktora

```
function Cat(name, age){
   this.name = name;
    this.age = age;
    this.sound = “Meeeeow!”;
}

Cat.prototype.makeSound = function(){
    console.log(this.sound);
}
Cat.prototype.speak = function(){
    console.log(‘Sorry cats can’t speak’);
}
```

ZADANIE 6

Stwórz funkcję konstruktora, która sprawia, że koty stają się obiektem od samego początku ze zdefiniowanym polami „**name**” oraz „**sound**” przekazanymi jako argument konstruktora.

Dodaj metodę „**makeSound**” która wyświetli w konsoli wartość pola „**sound**”, wywowałaj je dla nowo utworzonych kotów.

Tworzenie obiektów i dziedziczenie – Object.create()

Metoda Object.create () tworzy nowy obiekt, używając istniejącego obiektu w celu zapewnienia prototypu dla nowo utworzonego obiektu.

```
const base = {
        baseProperty: function(){
                console.log('I'm from prototype!')
        }
};

const obj = Object.create(base)

console.log(obj) // WTF? Empty object

obj.baseProperty() // I'm from prototype!
```

ZADANIE 7

Stwórz obiekt „cat”, który ma funkcję rejestrującą **this.sound**.

Stwórz nowy obiekt przez Object.create z obiektem „cat” podanym jako argument.

Rozszerz nowy obiekt o pole sound i wywołaj metodę która pokaże w konsoli **this.sound**.

Tworzenie obiektów i dziedziczenie - dziedziczenie

Dziedziczenie w JS oparte jest głównie na prototypach.

Ponieważ jednak funkcje konstruktora działają podobnie jak klasy, możemy również zbudować funkcję konstruktora, która utworzy obiekt oparty na innej funkcji konstruktora.

Tworzenie obiektów i dziedziczenie – funkcja konstruktora

```
function Animal(color, weight) {
      this.color = color;
      this.weight = weight;
}
Animal.prototype.getColor = function() { return this.color; }

function Cat(name, age, color, weight){
      Animal.apply(this, [color, weight]);
      this.name = name;
      this.age = age;
      this.sound = “Meeeeow!”;
}

const cat = new Cat(’Garfield’, 4, ’carrot’, ’8kg’)
console.log(cat.color, cat.weight, cat.getColor()) // ’carrot’, ’8kg’,
this.getColor is not a function
Cat.prototype = Object.create(Animal.prototype)
console.log(cat.getColor()) // ’carrot’
```

Autor:

ZADANIE 8

Stwórz funkcję konstruktora Animal o dowolnych właściwościach.

Stwórz funkcję konstruktora Dog o dowolnych właściwościach, która również rozszerza obiekty o pola z konstruktora Animal.

Stwórz obiekty psa i sprawdź w konsoli ich zawartość.

Autor:

Tworzenie obiektów i dziedziczenie – kompozycja vs. dziedziczenie

W JS istnieją dwa sposoby dziedziczenia, przez dziedziczenie prototypów oraz kompozycję obiektów

Tworzenie obiektów i dziedziczenie – kompozycja za pomocą Object.assign()

Metoda **Object.assign ()** służy do kopiowania wartości wszystkich własnych właściwości (właściwość własna - nie dziedziczona z **prototype**) z jednego lub więcej obiektów źródłowych do obiektu docelowego. Zwróci obiekt przekazany jako pierwszy argument.

```
const obj1 = { a: 1, b: 2, c: 3 }
const obj2 = { c: 4, d: 5 }
const obj3 = Object.assign(obj1, obj2)

console.log(obj3) // { a: 1, b: 2, c: 4, d: 5}
```

ZADANIE 9

Zrób 3 obiekty o losowych właściwościach (pola + metody).

Połącz je razem przez Object.assign, tworząc NOWY obiekt za pomocą kompozycji.

Tworzenie obiektów i dziedziczenie – metody i pola statyczne

- Z metod i pól statycznych można korzystać bez tworzenia obiektu (tzw. instancji) za pomocą funkcji konstruktora. Są one wywoływane bezpośrednio przez samą funkcję konstruktora (klasy).

- Metody statyczne nie mają bezpośredniego dostępu do metod i pól obiektów (instancji) utworzonych za pomocą funkcji konstruktora (klasy)

- Metody statyczne nie są dziedziczone przez obiekty (instancje) utworzone za pomocą funkcji konstruktora (klasy)

- Jeśli masz kod, który może być współużytkowany przez wszystkie instancje tej samej funkcji konstruktora (klasy), możesz wtedy umieścić tę część kodu w metodzie statycznej.

Tworzenie obiektów i dziedziczenie – metody i pola statyczne

Metoda instancji (obiektu utworzonego za pomocą funkcji) a metoda statyczna

- Metoda instancji ma bezpośredni dostęp do pozostałych metod instancji i zmiennych instancji.
- Metoda instancji może uzyskać bezpośredni dostęp do zmiennych statycznych i metod statycznych.

- Metody statyczne mogą uzyskiwać bezpośredni dostęp do zmiennych statycznych i pozostałych metod statycznych.
- Metody statyczne nie mają bezpośredniego dostępu do metod instancji i zmiennych instancji.

Tworzenie obiektów i dziedziczenie – metody i pola statyczne

```
function Dog(name, age) {
        Dog.count = Dog.count + 1;
        this.name = name;
        this.age = age;
}

Dog.count = 0;
Dog.showCount = function() {
        console.log(Dog.count);
}

Dog.showCount();
```

Tworzenie obiektów i dziedziczenie – Obiekty globalne

Pamiętajmy że tablice oraz funkcje w JS to również obiekty, lecz rozszerzone o specjalne pola i metody poprzez dziedziczenie prototypów z wbudowanych obiektów globalnych takich jak Object, Function, Array

```
function test() {}
console.log(typeof test === „object”) // true
// Function -> test
// Dostęp do odziedziczonych właściwości z globalnego obiektu Function
https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Function

const arrayExample = [];
console.log(typeof arrayExample === „object”) // true
// Array -> arrayExample
// Dostęp do odziedziczonych właściwości z globalnego obiektu Array
https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Array
```

https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Object

Tworzenie obiektów i dziedziczenie – Obiekty globalne

Wartości prymitywne typu „string” oraz „number”, w rzeczywistości nie są obiektami, lecz silnik JS w momencie wykonywania kodu i napotkania wartości prymitywnej tymczasowo ”opakowywuje” wartość prymitywną obiektem globalnym Number lub String dając nam możliwość do łatwego operowania danymi.

```
const test = ’test’;
console.log(typeof test === „string”) // true
// String -> test
// Dostęp do odziedziczonych właściwości z globalnego obiektu String
https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/String
```

```
const test = 34;
console.log(typeof test === „number”) // true
// Number-> test
// Dostęp do odziedziczonych właściwości z globalnego obiektu Number
https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Number
```

Tworzenie obiektów i dziedziczenie – Obiekty globalne

Dzięki temu możemy operować na wszystkich typach danych jak na zwykłych obiektach oraz korzystać z wielu gotowych funkcji oraz pól odziedziczonych po obiektach globalnych. Obiekty globalne to nic innego jak funkcje konstruktora.

```
let test = ‚hi hi’;
test = test.substr(1);

const test = [];
test.push(2)
test.forEach();

function test() {}
test.apply(this)
```