



this

Dostęp do pól obiektu

Obiekty są zazwyczaj tworzone po to, żeby pogrupować cechy wspólne np.:

```
var admin = {  
  name: "Janusz",  
  age: 67,  
  sayName: function () {  
    console.log("Janusz");  
  }  
};  
admin.sayName(); // Janusz
```

Obiekt admin posiada metodę **sayName**. Kiedy ją wywołamy **Janusz** do nas przemawia!

Dostęp do pól obiektu

A co gdybyśmy chcieli wypisać wartość, która istnieje pod kluczem **name** tego obiektu?

Jak w metodzie **sayName** wypisać wartość pola kluczowego **name**?

Można użyć nazwy obiektu, ale taki kod jest **niesolidny i nieużywalny**.

```
sayName: function () {  
    console.log(admin.name);  
}
```

Kontekst

Żeby dostać się do obiektu w metodzie używamy słowa kluczowego **this**.

```
sayName: function () {  
  console.log(this.name);  
}
```

this wskazuje na obiekt, w kontekście jakiego została wywołana metoda.

Pamiętaj o tym!!!!

this - przykłady

Ta sama funkcja może wskazywać na zupełnie różne **this** kiedy jest wywoływana z różnych obiektów.

```
var cat = {  
  name: "Filemon"  
};  
var dog = {  
  name: "Reksio"  
};  
  
function sayName() {  
  console.log(this.name);  
}  
  
//Przypisujemy funkcję do nowego  
//poła w obiektach  
cat.someF = sayName;  
dog.someF = sayName;  
cat.someF(); // Filemon  
dog.someF(); //Reksio
```

this - przykłady

Jeśli użyjemy **this** bez żadnego kontekstu, wskaże ono (w przypadku przeglądarki) **obiekt globalny Window**.

Jeśli wywołasz funkcję bez kontekstu to ta zawsze zwróci obiekt globalny.



Czas na zadania



Constructor function

Constructor function

Często potrzebujemy stworzyć wiele podobnych obiektów np. obiekty reprezentujące wielu użytkowników Twojej aplikacji.

Do stworzenia wielu podobnych obiektów możemy użyć funkcji, którą będziemy wywoływać poprzedzając słówkiem **new**.

Funkcję tą będziemy nazywać konstruktorem - **Constructor function**

```
function User() {  
    /* na razie pusto */  
}
```

Nazwę funkcji będącej konstruktorem piszemy z **dużej litery**! To tylko przyjęta zasada, aby odróżnić tę funkcję od innych.

Constructor function & new

Jeśli chcemy wywołać tę funkcję, tak aby stworzyła nam obiekt musimy poprzedzić jej wywołanie słówkiem kluczowym **new**.

Dzięki temu, że używamy słowa **new** funkcja User staje się konstruktorem. Konstruktor to rodzaj szablonu, który będzie opisywał nam jak mają wyglądać obiekty.

```
function User() {  
    this.name = "Ala";  
}  
var user1 = new User();  
var user2 = new User();  
var user3 = new User();  
console.log(user1, user2, user3);  
//User {name: "Ala"}  
//User {name: "Ala"}  
//User {name: "Ala"}
```

Na chwilę obecną wszyscy użytkownicy mają na imię Ala.

Constructor function & new

Jeśli chcielibyśmy stworzyć obiekty, które byłyby do siebie podobne, ale **różniły się** jakimiś polami musimy wywołać funkcję konstruktora i przekazać jej jakieś argumenty.

Dodajmy również dodatkowe pole **type**, które dla wszystkich obiektów typu User będzie takie samo.

```
function User(newName) {  
    this.name = newName;  
    this.type = "basic";  
}  
var user1 = new User("Ala");  
var user2 = new User("Janek");  
var user3 = new User("Bartek");  
console.log(user1, user2, user3);  
//User {name: "Ala", type: "basic"}  
//User {name: "Janek", type: "basic"}  
//User {name: "Bartek", type: "basic"}
```

Constructor function & new

Dodajmy również metodę do konstruktora i sprawdźmy w niej czym jest this.

```
function User(newName) {  
    this.name = newName;  
    this.type = "basic";  
    this.saySomething = function () {  
        console.log("Everyone knows all about my direction " + this.name);  
    }  
}  
  
var user1 = new User("Ala");  
var user2 = new User("Janek");  
var user3 = new User("Bartek");  
user1.saySomething(); // Everyone knows all about my direction Ala  
user2.saySomething(); // Everyone knows all about my direction Janek  
user3.saySomething(); // Everyone knows all about my direction Bartek
```

Constructor function & new - podsumowanie

- Funkcja, która tworzy obiekt z pomocą słowa kluczowego **new** to **konstruktor - Constructor function**;
- Konstruktory są to specjalne funkcje służące do stworzenia obiektu i ustawienia mu początkowego stanu;
- Konstruktor powinien być funkcją, którego nazwa zaczyna się **wielką literą**;
- Do ustawiania stanu w konstruktorze powinniśmy używać słowa kluczowego **this**;
- Aby stworzyć obiekt na bazie konstruktora, powinniśmy użyć słowa kluczowego **new**.

```
function Car(type, hp, color) {  
  this.type = type;  
  this.hp = hp;  
  this.color = color;  
};  
var fiat = new Car("fiat", 125, "red");  
console.log(fiat.type, fiat.hp, fiat.color);  
//fiat 125 red
```

Prototype

1000 użytkowników

W poprzednim rozdziale używaliśmy konstruktorów do tworzenia obiektów.

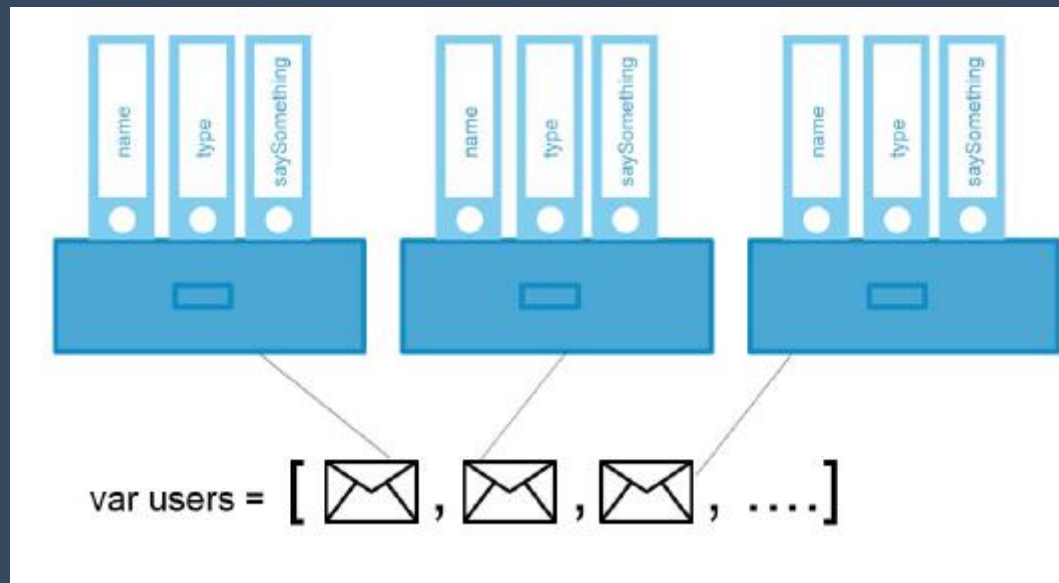
Dużą wadą tamtego rozwiązania jest fakt, że gdybyśmy chcieli stworzyć np. 1000 użytkowników dla naszej aplikacji, to każdy z nich zarezerwuje pamięć w przeglądarce dla takich samych, lub podobnych danych.

Na przykładzie obok stworzymy **tablicę takich samych obiektów**. Ich podobieństwo polega na tym, że mają takie same pola oraz ich metoda robi to samo.

```
function User(newName) {  
  this.name = newName;  
  this.type = "basic";  
  this.saySomething = function () {  
    console.log("Hi" + this.name);  
  }  
}  
  
var users = [];  
for (var i = 0; i < 1000; i++) {  
  users.push(new User("user" + i));  
}
```

1000 użytkowników

Spójrz, funkcja `saySomething` oraz wartość `type` są takie same dla każdego użytkownika. A może dałoby się je wydzielić do jakiegoś jednego, wspólnego obszaru, żeby niepotrzebnie nie tworzyć ich 1000 razy?



1000 użytkowników

Zanim zajmiemy się naszym problemem z poprzednich slajdów, potrzebujemy trochę więcej wiedzy o obiektach.

Teraz kiedy już wiesz jak wyglądają obiekty, ważne żeby sobie zdać sprawę z tego, co jest obiektem, a co nie.

Oprócz typów prostych (`string`, `number`, `boolean`, `null`, `undefined` i `symbol`) **wszystko inne w JavaScript jest obiektem**. To znaczy, że tablice i funkcje również są obiektami.

Skoro funkcje są obiektami to oznacza, że mamy dostęp do wielu ciekawych właściwości.

F.prototype – co to jest?

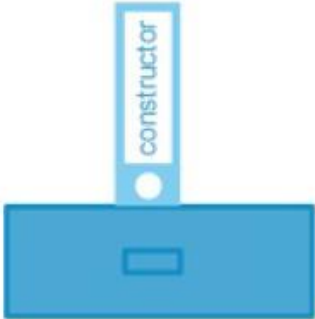
Każda funkcja w JavaScript posiada specjalne pole o nazwie **prototype**.

Spróbujmy stworzyć pustą funkcję i przyjrzeć się jej z bliska.

Jak widzisz obok, element, do którego się dostajemy poprzez pole **prototype** funkcji to obiekt z polem **constructor**. Na razie nie będziemy zajmować się tym polem, ani innymi tu stworzonymi. Będziemy dodawać własne pola do tego obiektu.

```
function test() {}  
console.dir(test);
```



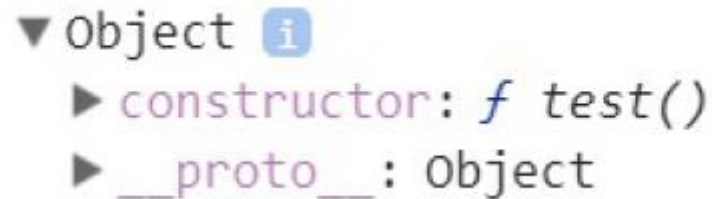
```
f test()  
  arguments: null  
  caller: null  
  length: 0  
  name: "test"  
  ▶ prototype:  →   
  ▶ __proto__: f ()  
    [[FunctionLocation]]: VM20772:1  
  ▶ [[Scopes]]: Scopes[2]
```

F.prototype – jest obiektem

Dostajemy się do tego obiektu wpisując nazwę funkcji, a po kropce nazwę pola czyli **prototype**.

```
function test() {}  
console.dir(test.prototype);
```

Obiekt **test.prototype** jest widoczny dla każdego obiektu, który stworzymy na podstawie funkcji - jeśli będzie ona oczywiście konstruktorem.



```
▼ Object ⓘ  
  ► constructor: f test()  
  ► __proto__: Object
```

User.prototype – dodawanie nowych pól

Stwórzmy zatem funkcję będącą konstruktorem i spróbujmy dopisać nowe pole do jego obiektu pod kluczem **prototype**.

Obiekt ten jest widoczny dla każdego obiektu, który stworzymy na podstawie funkcji - jeśli będzie ona oczywiście konstruktorem, więc spróbujmy również dostać się do nowego pola poprzez obiekty.

Spójrz poniżej. Z przykładu, który omawialiśmy wcześniej zabraliśmy pole **type** z konstruktora. Dołożone ono zostało do obiektu dostępnego poprzez pole **prototype**. Jak widzisz wszystkie obiekty mają do niego dostęp.

```
function User(newName) {  
  this.name = newName;  
  this.saySomething = function () {  
    console.log("Hi" + this.name);  
  }  
}  
User.prototype.type = "basic";  
var user1 = new User("Ala");  
var user2 = new User("Janek");  
console.log(user1.type, user2.type);  
//basic basic
```

User.prototype – dodawanie nowych pól

Wyciągnijmy również metodę **SaySomething**.

```
function User(newName) {  
    this.name = newName;  
}  
User.prototype.type = "basic";  
User.prototype.saySomething = function () {  
    console.log("Hi " + this.name);  
}  
var user1 = new User("Ala");  
var user2 = new User("Janek");  
console.log(user1.saySomething()); //Hi Ala  
console.log(user2.saySomething()); //Hi Janek
```

Wszystkie obiekty, które są stworzone na bazie konstruktora **User** mogą korzystać z obiektu dostępnego poprzez **prototype**. Pytanie w jaki sposób?

__proto__ - co to jest?

Każdy obiekt w JavaScript, KAŻDY, posiada specjalne pole, które w większości przeglądarek np. Chrome, jest wyświetlane pod nazwą **__proto__**.

Wymawiamy jako "dunder proto" czyli skrót od "double underscore proto".

Pole **__proto__** to również odniesienie do innego obiektu, podobnie jak pole prototype w funkcjach (Funkcje jako, że też są obiektami również mają pole **__proto__**).

Tworzymy obiekt **cat** z jednym polem **name**. Kiedy wyświetlimy obiekt w konsoli widać, że rzeczywiście ma jeszcze jedno pole o nazwie **__proto__**

```
> var cat = { name: "Filemon"}  
< undefined  
  
> cat  
< ▼ {name: "Filemon"} ⓘ  
  name: "Filemon"  
  ► __proto__: Object  
  
> |
```

`__proto__` - ustawianie, nie wskazany przykład

Każdy obiekt ma swój prototyp w postaci innego obiektu, z którego dziedziczy pewne właściwości.

Prototyp obiektu jest dostępny poprzez pole `__proto__`, jest to ukryta wewnętrzna własność obiektu.

Łańcuch tak połączonych obiektów jest nazywany łańcuchem prototypu - **prototype chain**.

Jeżeli próbujemy odwołać się do pola w obiekcie, którego nie ma, JavaScript zaczyna przeszukiwanie połączonych z nim obiektów, połączonych właśnie przez pole `__proto__`.

```
var animal = {  
  isBreathing: true  
};  
var horse = {  
  name: "Karino",  
  __proto__: animal  
};  
console.log(horse.isBreathing); //true
```

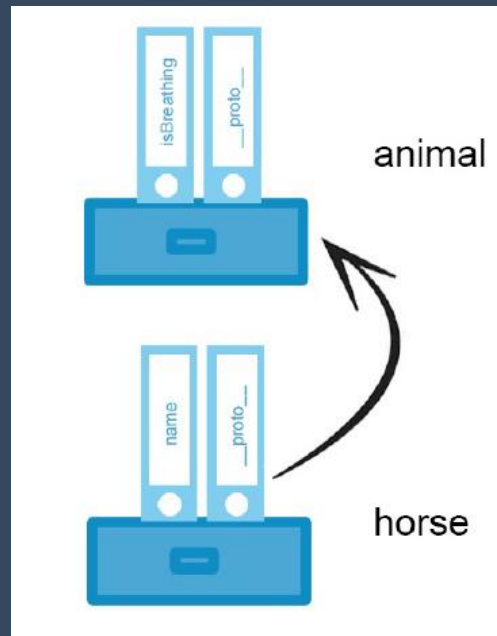
Nie rób nigdy takiego połączenia. To tylko przykład, aby zrozumieć połączenie.

`__proto__`

Na obrazku poniżej mamy graficznie przedstawiony (skrócony) łańcuch prototypu.

Nie powinno się używać w kodzie bezpośrednio słowa `__proto__`, ponieważ nie jest ono we wszystkich przeglądarkach tak samo zaimplementowane.

Aby ustawiać połączenia między obiektami używamy innych sposobów np. słówka `new`.



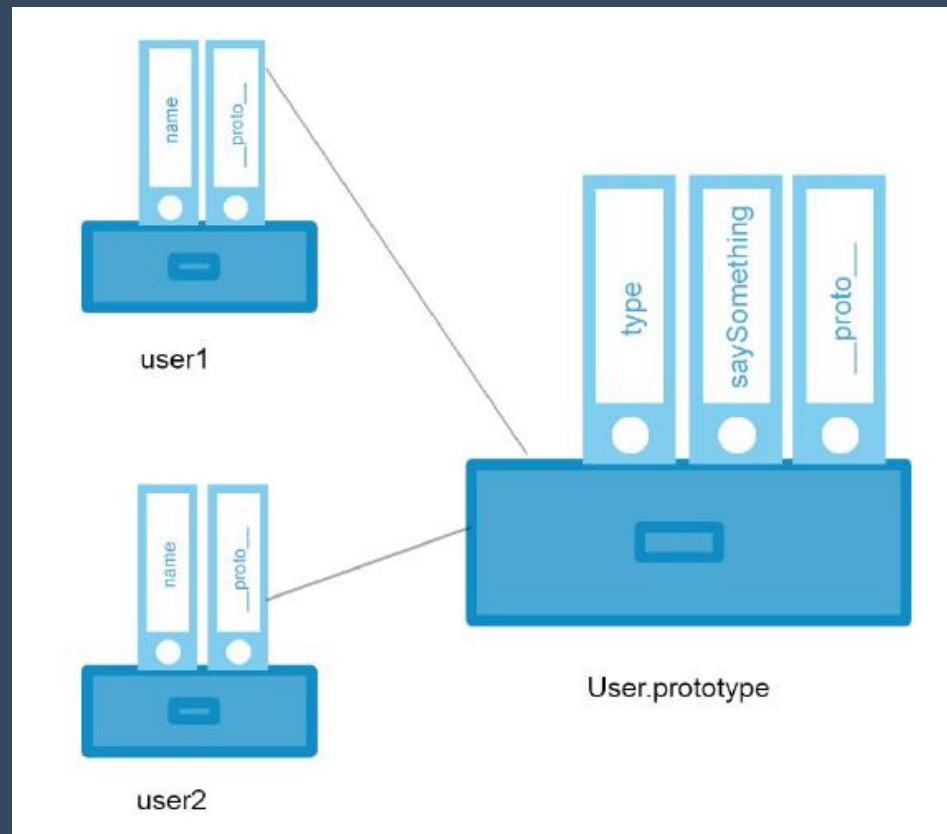
Co robi new? – najważniejsze rzeczy

1. Tworzy nowy pusty obiekt {}
2. Ustawia pole `__proto__` nowego obiektu tak, aby wskazywało na pole `prototype` konstruktora
3. Ustawia `this` tak, aby wskazywało na nowo powstały obiekt
4. Zwraca nowo powstały obiekt.

```
function User(newName) {  
  //1. var nowy = {}  
  //2. nowy.__proto__ = User.prototype  
  //3. this = nowy  
  this.name = newName;  
  //4. return this  
}  
User.prototype.type = "basic";  
User.prototype.saySomething = function () {  
  console.log("Hi " + this.name);  
}  
var user1 = new User("Ala");
```

__proto__ vs [[Prototype]]

Na obrazku poniżej widzisz jak zostały połączone obiekty user1 oraz user2 z obiektem User.prototype za pomocą new



Podsumowanie

Pamiętaj nigdy nie używaj w swoim kodzie bezpośrednio słowa `__proto__`, to zła praktyka, ze względu na różną implementację dostępu do łańcucha prototypów.

Temat prototypów nie jest taki trudny jak mogłoby się wydawać. Spróbuj wykonać zadania, a na pewno trochę lepiej zrozumiesz temat.



Czas na zadania