Stary dziadek czy młodszy brat rubiego?
Czyli javascript jako nowoczesny i pełnoprawny język programowania.

Autor: Jacek Młynek
e-mail: jacek.javascript@gmail.com

Dlaczego tak o js?

Pytanie.
Dlaczego javascript jest zarazem najczęściej wykorzystywany i jednocześnie najbardziej niezrozumiałym językiem na Świecie?

Dlaczego ?

- Tak często programiści nawet doświadczeni stosują złe praktyki w js:
  - Zmienne globalne.
  - Funkcje po 100 i więcej linii kodu.
  - Zero przestrzegania Single Responsibility Principle.
- Tak często pragną „ulepszać” js:
  - Modelowanie języków statycznie typowanych.
  - Próby implementacji klas, interfejsów.

Odpowiedz:

Okazuję się że dla nich spełnione jest tylko równanie:
js == językProgramowania //true

Natomiast równanie:

js === językProgramowania //false

Jak to można zmienić?

Najlepiej pokazać coś nowego odwołując się do czegoś znanego.

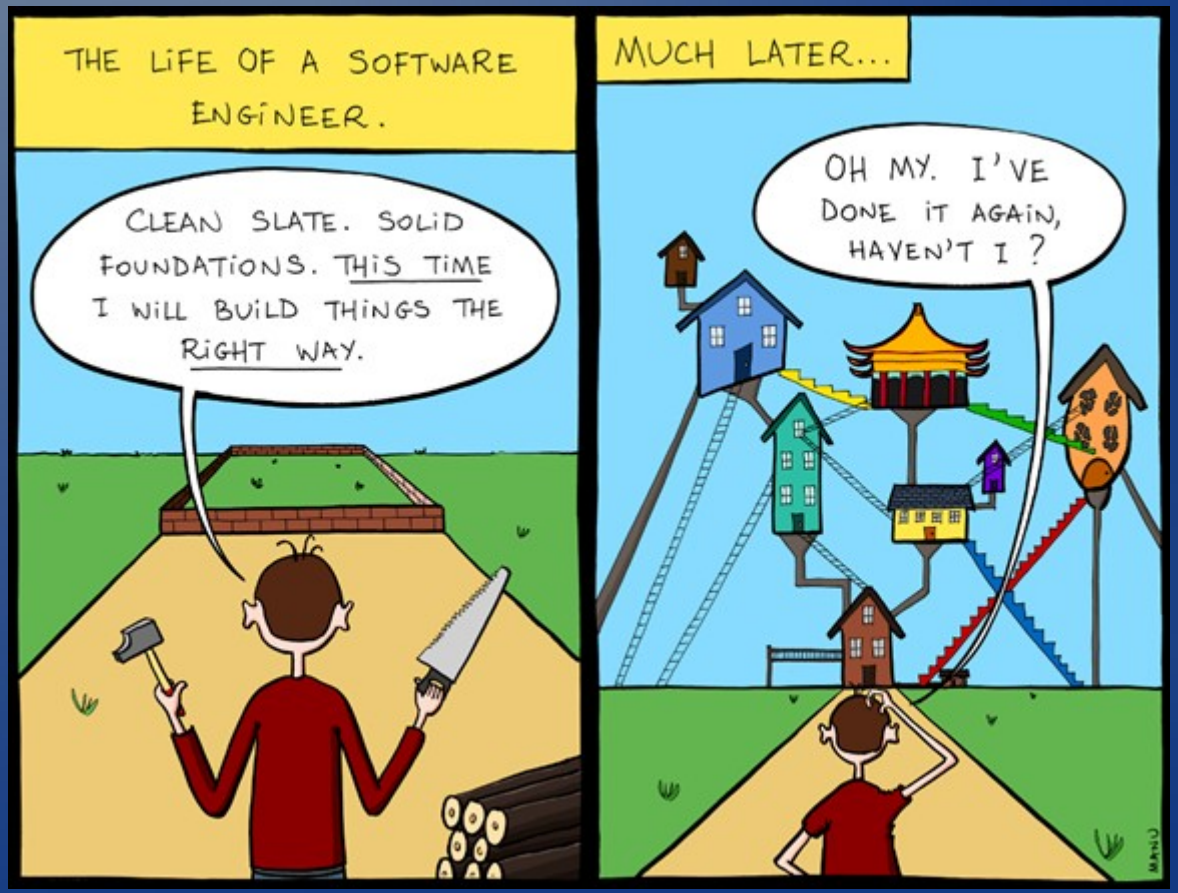
Odwołajmy się zatem do doświadczeń wielu lat programistów i sięgnijmy po wzorce projektowe.

Sięgnijmy po jeden z obecnie najchętniej wybieranych języków, rubiego i pokażemy że js mimo iż starszy dotrzymuje mu kroku.

Główny cel prezentacji:

Pobudzić Was do bycia architektami własnych systemów w js, nie tylko Bobem budowniczym, Maliniakiem określonej struktury.

Żebyście to Wy wybierali w czym wam najlepiej a nie moda, biblioteka wybierała za Was.



Konkrety Panie kolego !

Proste jest piękne !

- Doświadczenia kilku pokoleń programistów uczą że im mniej coś skomplikowane tym lepiej.
- Javasript jest prostym językiem:
  - Jest luźno typowany (duck typing).
  - Ma tylko kilka typów wbudowanych.
  - Prosta struktura obiektu (na nim wzorowano json).
  - Brak klas.
  - Lambda.

proste === jestPiekne
&&
prostyJezykProgramowania
!== nieMuszeSieGoUczyc;

Czyli kilka ważnych rzeczy o js.

Jeśli kiedyś powstanie Kościół js to początek będzie miał w funkcji.

- To też obiekt i ma swoje metody np.: apply.
- Ogranicza dostęp do zmiennych zdefiniowanych w jej ciele.
- W trakcie wywołanie przekazujemy dwa ukryte parametry (this, arguments).
- Każda funkcja ma własny prototyp i każda dziedziczy po Function.prototype.
- Tworzy domknięcie (binduje zmienne lokalne do środowiska definicji a nie wykonania).

Javascript w akcji czyli jak szukałem klasy.

Przed …symulowałem

```
var Car = function(name){
    this.name = name;
}

Car.prototype.drive = function(){
    console.log("im driving");

}
var myCar = new Car(name);

var SportsCar = function() {
    //repeat constrouctor is the only way.
    this.name;
}

//tak jak w klasycznym progamownaiu pozwala wywolac metode klasy bazowej.
SportsCar.inherits(Car);

SportsCar.prototype.drive = function() {
    if(this.name) {
        return this.super(drive);
    }

    return "0";
}
```

Przed … pytałem; Closure czy prototype?

Closure:

- Zmienne prywatne
- Wolniej działa

Prototype:

- Wszystko publiczne
- Nie powtarza kodu jest szybszy zatem.

Punkt zwrotny !

- Co dają mi klasy:
  - Dają początek grupie obiektów.
  - Dostarczają dodatkowe miejsce na definicje metod i zmiennych statycznych.
- Co narzucają przy modelowaniu domeny:
  - Źle rozumiany DRY prowadzić może do złego dziedziczenia klasowego (Klasa domenowa dziedziczy po klasie narzędziowej).
  - Powinny przestrzegać SRP.
  - Mocne powiązania w relacji dziedziczenia. (Liskov substitution principle).

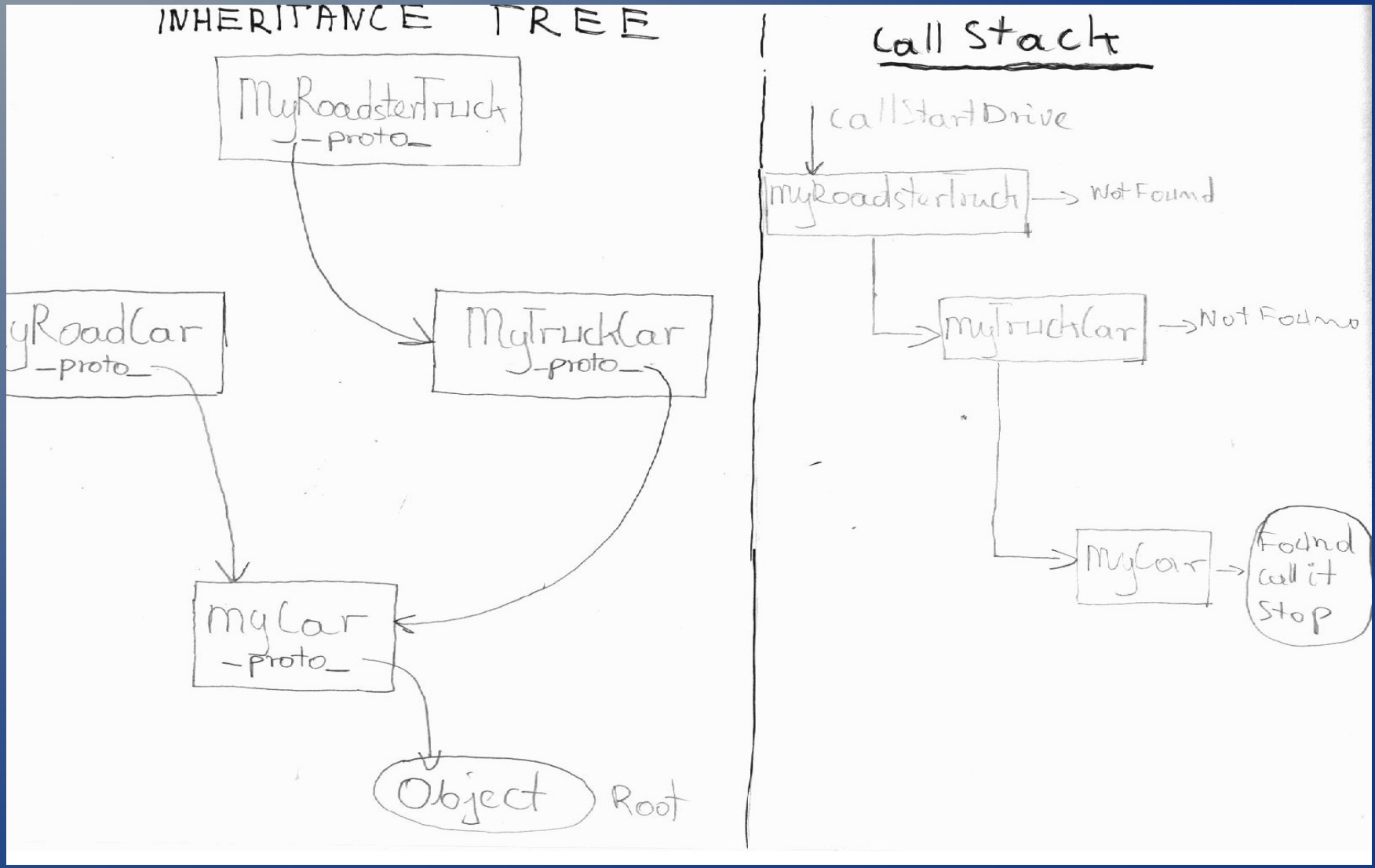
Twórca java'y zapytany po latach czy jest coś co zmieniłby w swoim języku odparł że pozbył by się klas.

Ok. Ale czy obiekt może istnieć bez klasy?
W js jak najbardziej.
Więcej,
nawet jest to naturalniejsze, niż próba ich modelowania.

Js zaczyna nabierać kolorów

Zapomnijmy na razie o tym co wiemy o tworzeniu obiektów.
Wróćmy do tego na czym twórcy języków się wzorowali.
Do przyrody.
Czy w przyrodzie osobnik X powstaje z jakieś wydmuszki(klasa) innego osobnika?
Nie.
Powstaje z innego w pełni „funkcjonalnego”.

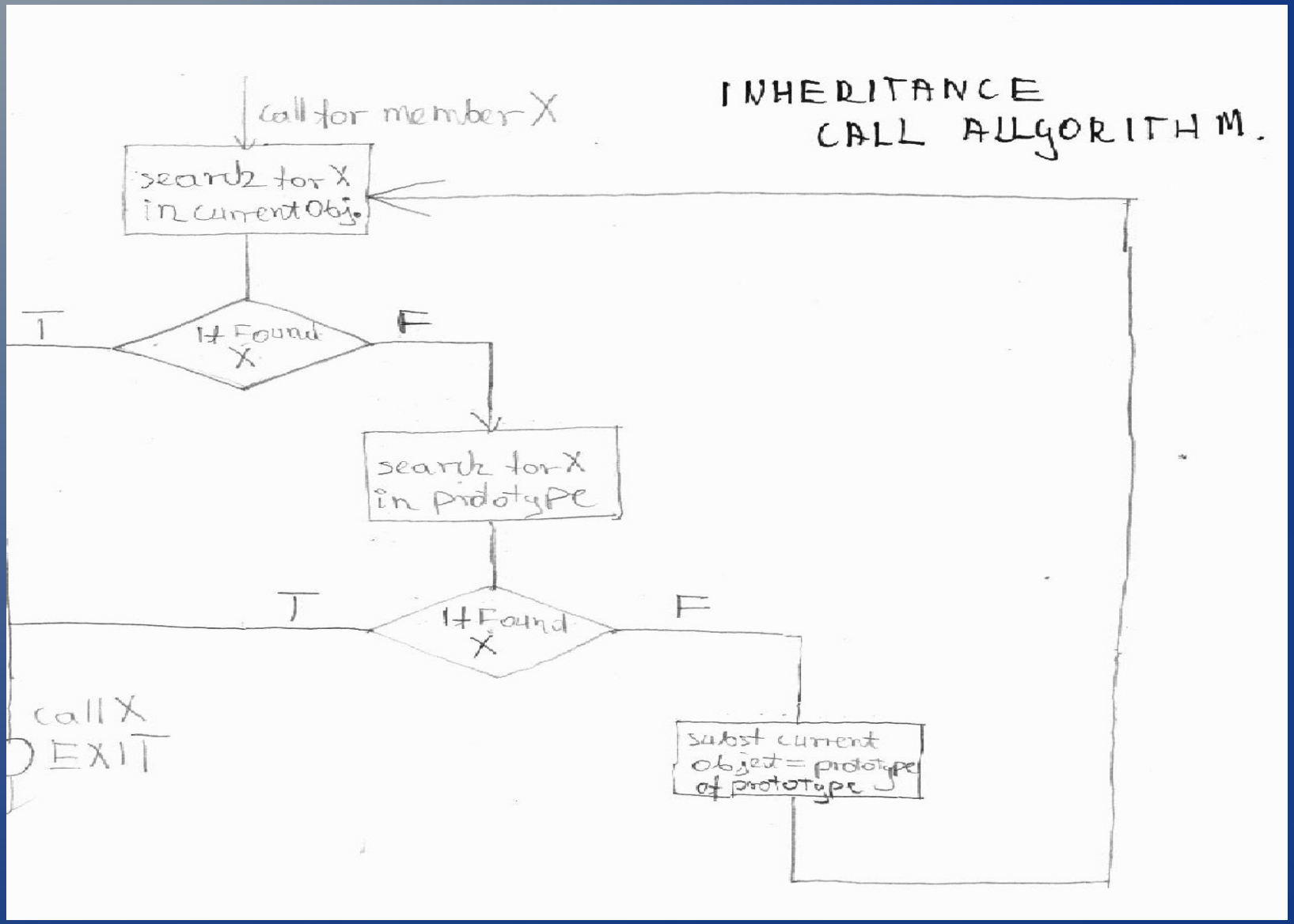
Przykład – samochód jako prototype.js



Przykład – samochód jako prototype.js

```
var myTruckCar = Object.create(myCar);

//Differances only
myTruckCar.name = "My track has become truck prototype ";
myTruckCar.sound = "hrrrrr plum";

myTruckCar.containerType = "long";
myTruckCar.cargoType = "wood";
myTruckCar.relaseCargo = function (){
  console.log("cargo  has been relased");
};

myTruckCar.startDrive();
myTruckCar.relaseCargo();

// aguments params.
var myRoadsterTruck = Object.create(myTruckCar,{
  name: {
      value: "My beautiful truck",
      writable: true,
      configurable: true,
      enumerable: true},
  containerType: {value: "short"}
  });

myRoadsterTruck.startDrive(); // it has not have its own sound so shoud ge

// It will be not needed i my shiny new truck :)
myRoadsterTruck.cargoType = undefined;
myRoadsterTruck.relaseCargo = undefined;

//myRoadsterTruck.relaseCargo(); //throw error;
```

Prosty zapis kodu genetycznego.



Co nam to wnosi?

- Prostsze, naturalniejsze tworzenie obiektów, w oparciu o w pełni działający obiekt nie wydmuszkę.
- Nie łamiemy SRP, obiekt może w trakcie swojego działania robić różne rzeczy.
- Proste dziedziczenie.

No dobrze, ale czy to rozwiązuje wszystkie problem dziedziczenia?

Śladem rubiego w js.

Delegacja zamiast dziedziczenia lekarstwem na DRY. Cz.1

Ruby

```ruby
object_to_extend = Object.new
not_extented_object = Object.new

object_to_extend.instance_eval do
  def say_something
    puts 'somthing on object to extend'
  end
end
```

js

```js
(function(){
  var objectToExtend = {};
  var notExtended_object = {};

  objectToExtend.say_something = function ()
  {
    console.log("something from objectToExtend");
  }
```

Delegacja zamiast dziedziczenia lekarstwem na DRY. Cz.2

- Ruby

```ruby
module Printer
  def print_to_pdf
    puts "Text as pdf: #{self.details}"
  end

  def print_to_html
    puts "Text as html: #{self.details}"
  end

  def print_to_csv
    puts "Text as csv: #{self.details}"
  end
end

class Offer
  attr_accessor :details
end

saving_account_offer = Offer.new
saving_account_offer.extend Printer
saving_account_offer.details = "Nice saving account with i
saving_account_offer.print_to_html if saving_account_offer
```

- js

```javascript
var printer = {
  printToPdf: {
    value: function()
    {
      console.log("Text as pdf: " + this.details);
    }
  },
  printToHtml:{
    value: function()
    {
      console.log("Text as html: " + this.details);
    }
  },
  printToCsv: {
    value: function(){
      console.log("Text as csv: " + this.details);
    }
  }
}

savingAccountOffer = {
  details: "Nice saving account with intrest rate up to
},

//Using ECMAScript 5
Object.defineProperties(savingAccountOffer, printer);

//Using my coustom extend
//Object.extend(savingAccountOffer, printer);

savingAccountOffer.printToPdf();
```

Mixin.js

Ruby

```ruby
module Printer
  def print_to_pdf
    puts "Text as pdf: #{self.details}"
  end

  def print_to_html
    puts "Text as html: #{self.details}"
  end

  def print_to_csv
    puts "Text as csv: #{self.details}"
  end
end

class Invoice
  include Printer
  attr_accessor :details
end

tv_invoice = Invoice.new
tv_invoice.details = "tv invoice for march equal 1
tv_invoice.print_to_pdf if tv_invoice.respond_to?
```

js

```javascript
var printer = {
  printToPdf: {
    value: function()
    {
      console.log("Text as pdf: " + this.details);
    }
  },
  printToHtml: {
    value: function()
    {
      console.log("Text as html: " + this.details);
    }
  },
  printToCsv: {
    value: function(){
      console.log("Text as csv: " + this.details);
    }
  }
};

var invoiceMaker = function()
{
  var _details;
  var _proto = {};
  Object.defineProperties(_proto, printer);
  var _that = Object.create(_proto);

  _that.details = _details;

  return _that;
};

tvInvoice = invoiceMaker();
tvInvoice.details = "tv invoice for march equal 120
if(tvInvoice.printToCsv !== undefined) tvInvoice.pri
```

Ehh, jeszcze gdyby się dało zachować hermetyzację niektórych danych.

Domknięcie/Block/Closure.

Podobnie jak ruby js posiada możliwość zbindowania zmiennych lokalnych ze środowiskiem w którym zostały zdefiniowane a nie ze środowiskiem w jakim są uruchamiane. Innymi słowy js posiada domknięcie blok w rubim.

Domknięcie przykład.

Ruby

```ruby
index = 123

puts "index before changes #{index}"

def change_in_local_index
 #puts "index just before" + index
  index = 1
  lambda { 3.times {puts index +=1 } }
end

change_in_local_index.call

puts "index after changes #{index}"
```

js

```javascript
(function(){
  console.log("Access local variable example");
  var _index = 123;
  console.log("index before change: " + _index);

  function change_in_local_index()
  {
    var _index = 1;
    return function(){
      for(var i=0; i<3; i++)
      {
        console.log(_index+=1);
      }
    }();
  }

  change_in_local_index();

  console.log("index after changes: " + _index);

})();
```

Praktyczne wykorzystanie domknięcia:

- Iteratory:
  - Sortowanie
  - Filtrowanie
  - Predykaty
- Ukrywanie implementacji:
  - Tworzenie obiektów ze zmiennymi i metodami prywatnymi

Przepis na ukrywanie implementacji - opis.

1. Tworzymy funkcje owrapującą.
2. Tworzymy za pomocą Object.create lub przekazujemy obiekt prototypu jeśli chcemy zastosować dziedziczenie.
3. Tworzymy zmienne i metody prywatne.
4. Tworzymy metody, publiczne
5. Zwracamy obiekt.

Przepis na ukrywanie implementacji - przykład.

```
(function() {
  var myCarMaker = function(spec){
    var _sound = spec.sound || "brummm";
    var _proto = spec.proto || {};

    //Create new object
    var _that = Object.create(_proto);

    // make private methods
    var _startEngin = function(){
      console.log("StartEngin");
    };

    var _shiftGear = function(){
      console.log("Shift gear");
    };

    var _pressAccelerator = function(){
      console.log("Press accelerator and brummm");
    };

    //Privilages methods
    _that.get_name = function () {return spec.name;};
    _that.startDrive = function ()
    {
      console.log(spec.name);
      _startEngin();
      console.log(_sound);
      _shiftGear();
      _pressAccelerator();
    };

    return _that;
  };

  var myCar = myCarMaker({
    name: "My private car become prototype",
    sound: "trtr"
  });
```

Podsumujmy.

- Podobnie jak w ruby możemy stosować domknięcia, przydatne zwłaszcza w ukrywaniu implementacji i iteratorach.
- Możemy dowolni rozszerzać obiekty
- Możemy tworzyć obiekty.
- Możemy dziedziczyć.
- Możemy w końcu to wszystko osiągnąć bez udziału klas czego nie możemy zrobić w rubim.

Wnioski?

Javascript mimo swojego wieku wydaje się być całkiem rześkim dziadkiem.

Może się czasami potknie (przekazywanie this, czy też operator new), nie mniej w niektórych momentach jest wstanie prześcignąć rubiego (brak klas, prostota).

Wydaje się zatem idealnym językiem do modelowania domeny.

Literatura:

- JavaScript: The Definitive Guide, 5th Edition, David Flanagan
- JavaScript: The Good Parts, Douglas Crockford
- http://javascript.crockford.com/prototypal.html

Coming soon with:
„Data, context and interaction in js”

Dziękuje :O)