API

Aplikacijų programavimo sąsaja (angl. Application Programming Interface, API) – tai sąsaja, kurią suteikia kompiuterinė sistema, biblioteka ar programa tam, kad programuotojas per kitą programą galėtų pasiekti jos funkcionalumą ar apsikeistų su ja duomenimis.

In computer programming, an application programming interface (API) is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components. A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer. An API may be for a web-based system, operating system, database system, computer hardware or software library. An API specification can take many forms, but often includes specifications for routines, data structures, object classes, variables or remote calls. POSIX, Microsoft Windows API, the C++ Standard Template Library and Java APIs are examples of different forms of APIs. Documentation for the API is usually provided to facilitate usage.

JavaScript objektai

JavaScript kalboje dauguma dalykų yra objektai, nuo eilučių ir masyvų iki naršyklės **API**. Dažniausiai susijusios funkcijos ir kintamieji yra patalpinami į objektus, kurie tampa patogiais duomenų konteineriais.

Pavyzdys, kaip galime sukurti naują objektą su {} skliaustais. Jis turi keletą išsaugotus kintamuosius ir vieną funkciją.

```
var car = {
    name: "Volvo",
    model: "S90",
    year: "2005",
    print: function() {
        console.log(this.name + " " + this.model + ". Production date: " +
this.year + ".");
    }
}

car.print();
```

JavaScript kalba yra keletos paradigmų (objektinė, funkcinė). O su objektais ir jų paveldimimu galime nuveikti daug daugiau nei tik sukurti naują objektą.

Vienas iš JavaScript dinaminės kalbos privalumų yra tas, kad gali objektų struktūrą, po to kai jie jau buvo sukurti. Pridėkite dar vieną savybė car objektui:

```
car.mileage = 180000;
console.log(car);
```

Bet kaip mes galime kurti visą aplikaciją, kai objektai gali keistis, o aš tikiuosi nustatytos struktūros.

JavaScript kalba taip pat gali kurti objektus, kurie turės griežčiau apibrėžtą struktūrą. O naujus objektus kursime būtent tokios struktūros, tik su skirtingais duomenimis. Šis būdas labiau primena *class* griežtai tipizuotose kalbose, bet nėra atitikmuo.

Tarkime mes norime sukurti kažką, kas leistų kurti mums keletą *car* objektų, bet su ta pačia struktūra. Tai mes galime padaryti su *new* žodžiu.

```
function Car() {
    this.name = "Volvo";
    this.model = "S90";
    this.year = 2005;
    this.print = function() {
        console.log(this.name + " " + this.model + ". Production date: " +
this.year + ".");
    }
}

var volvo = new Car();
```

Čia matome paprastą funkciją *Car()*, kuri priskiria priskiria reikšmes objektui pateiktam kaip *this.* Žodelis *this* JavaScript kalboje siejamas su objektu. Tai bet kuris objektas, kuris vykdo dabartinį programos kodą. Pagal nutylėjimą, tai globalus objektas, pavyzdžiui naršyklėje tai bus *Window* objektas.

Kai mes vykdėme *Car()* funkciją, kas buvo *this*? Kadangi mes naudojome žodelį *new,* tai buvo naujas tuščias objektas. Ką padaro žodelis *new*, tai sukuria tuščia objektą ir priskiria jo kontekstą žodeliui *this* ir pakviečia *Car()* funkciją.

Užduotis: Kas būtų, jei pakviestumėm *Car()* funkciją be *new* žodelio?

Mes taip pat nenorime, kad visos mašinos būtų volvo, tad perduokime reikšmes funkcijai.

```
function Car(name, model, year) {
    this.name = name,
    this.model = model,
    this.year = year,
    this.print = function() {
        console.log(this.name + " " + this.model + ". Production date: " +
this.year + ".");
    }
}

var volvo = new Car("Volvo", "S90", 2005);
```

Tokio tipo funkcijos, kaip Car(), kurios naudojamos kurti objektams yra vadinamos **konstruktorių funkcijomis.** Tai paprastos funkcijos, tačiau JavaScript kalboje tai labai dažnas būdas, kaip kurti objektus.

Apžvelgėme du būdus, kaip sukurti objektus, tačiau tai tik sintaksės pagražinimas (syntactic sugar). Abu būdai naudoja vieną ir tą patį *Object.create* metodą.

```
var volvo = Object.create(Object.prototype,
{
    name: {
        value: "Volvo",
        enumerable: true,
        writable: true,
        configurable: true
    },
    model: {
        value: "S90",
        enumerable: true,
        writable: true,
        configurable: true
    },
    year: {
```

```
        value: 2005,
        enumerable: true,
        writable: true,
        configurable: true
    }
});
```

ES6 ir objektai

Naršyklės, kurios palaiko ES6, turi naują žodelį *class,* kuris veikimu primene tipizuotas kalbas. Bet tai tik syntactic sugar jau esamiems objektų kūrimo būdams.

```
class Car {
    constructor(name, model, year){
        this.name = name,
        this.model = model,
        this.year = year
    }

    print() {
        console.log(this.name + " " + this.model + ". Production date: " +
this.year + ".");
    }
}

var volvo = new Car("Volvo", "S90", 2005);
volvo.print();
```

Labai panaši į prieš tai rašytą konstrutoriaus funkciją, tačiau šis variantas kai kuriems gali būti labiau įprastas klasės aprašymas.

Object savybės

Objektų savybes galime pasiekti keliais būdais, su tašku arba laužtiniais skliaustais:

```
var car = {
    name: "Volvo",
    model: "S90",
    year: "2005"
}

console.log(car.name);
console.log(car["name"]);
```

Tačiau objektų savybes sudaro daugiau negu tik key value poros (pavadinimai ir reikšmės).

```
var car = {
    name: "Volvo",
    model: "S90",
    year: "2005"
}

console.log(Object.getOwnPropertyDescriptor(car, "name"));
```

Dar matome papildomus 3 atributus:

- Configurable (galima uždrausti property keitimą ir trynimą),
- Enumerable (ar property gali būti pasiekiama naudojant ciklą. Taip pat jei nustatomas false, serializuojant duomenis į JSON (JSON.stringify()), savybė bus neįtraukta),
- Writable (ar value gali būti pakeistas ar jis **read only**)