ობიექტზე ორიენტირებული პროგრამირება

მაგალითებში სადაც Var შეგვხვდება ვანაცვლებთ let-ით.

ობიექტები

დღისათვის ობიექტზე ორიენტირებული პროგრამირება წარმოადგენს გაბატონებულ პარადიგმას პროგრამული უზრუნველყოფის შექმნის პროცესში. JavaScript-ში ჩვენ შეგვიძლია გამოვიყენოთ ყველა ის უპირატესობა, რასაც გვაძლევს OOP (Object-oriented programming). მაგრამ ჯავასკრიპტისთვის ობიექტზე ორიენტირებულ პროგრამირებას გააჩნია თავისი ნიუანსები.

პროგრამირების ობიექტზე ორიენტირებულ კლასიკურ ენებში, როგორიცაა C++, Java, C#, OOP-is მთელი კონცეფცია კლასისა და ობიექტის ცნებების ირგვლივ ტრიალებს. კლასი წარმოადგენს ობიექტის ერთგვარ იდეალურ გეგმას, ხოლო ობიექტი - ამ გეგმის კონკრეტულ განხორციელებას. მაგალითად, ყველას აქვს წარმოდგენა სახლზე, როგორც საგანზე, რომელსაც აქვს კედლები და სახურავი. ანუ ეს წარმოდგენა არის რაღაც იდეალური გეგმა, რომელიც შეიძლება წარმოვიდგინოთ, როგორც კლასი. ამავდროულად არსებობს კონკრეტული სახლები, რომლებიც ერთმანეთისგან განსხვავდება მასალით, რისგანაც არიან აგებულნი, სართულების რაოდენობით, ფანჯრებით, ფორმით და ასე შემდეგ. ანუ სახლი გამოდის კლასის როლსი, სოლო კონკრეტული სახლები ამ კლასის ობიექტებია.

JavaScript-ში ECMAScript 6 სტანდარტამდე არ არსებობდა კლასის ცნება, მის ნაცვლად გამოიყენებოდა ობიექტის პროტოტიპი. რადგანაც ECMAScript 6 სტანდარტი ჯერ კიდევ შორსაა ბრაუზერულ JavaScript-ში სრული დანერგვისაგან, ჩვენ განვიხილავთ პროტოტიპების გამოყენებას.

ობიექტები

წინა თემებში ჩვენ ვმუშაობდით პრმიტიულ მონაცემებთან - რიცხვებთან, სტრიქონებთან, მაგრამ მონაცემები ყოველთვის არაა წარმოდგენილი მხოლოდ პრიმიტიულ ტიპებში. მაგალითად, თუ ჩვენს პროგრამაში გვჭირდება ადამიანის არსის აღწერა, რომელსაც აქვს სახელი, ასაკი, სქესი და ასე შემდეგ, მისი აღწერისთვის პრიმიტიული ტიპები საკმარისი არაა - ადამიანის არსს ვერ აღვწერთ მარტო რიცხვით ან მარტო სტრიქონით, საჭიროა ბევრი რიცხვი და სტრიქონი. ამდენად, ადამიანი წარმოადგენს კომპლექსურ

სტრუქტურას, რომელსაც გააჩნია თვისებები: სახელი, გვარი, ასაკი, სქესი, პროფესია და ასე შემდეგ.

ასეთ სტრუქტურებთან სამუშაოდ JavaScript-ში გამოიყენება ობიექტები. თითოეული ობიექტი შეიძლება შეიცავდეს თვისებებს, რომლებიც აღწერს მის მდგომარეობას და მეთოდებს, რომლებიც აღწერენ მის ქცევას.

ახალი ობიექტის შექმნა

არსებობს ახალი ობიექტის შექმნის რამდენიმე ხერხი.

პირველი ხერხი მდგომარეობს object-ის კონსტრუქტორის გამოყენებაში:

```
Let user = new Object();
```

ამ შემთხვევაში ობიექტს ჰქვის user. ის განისაზღვრება ისევე, როგორც სხვა ცვლადები საკვანძო let სიტყვის მეშვეობით.

გამოსახულება new Object() წარმოადგენს კონსტრუქტორის გამოძახებას, ანუ გამოძახებას ფუნქციისა, რომელიც ქმნის ახალ ობიექტს. კონსტრუქტორის გამოსაძახებლად გამოიყენება ოპერატორი new.

ობიექტის შექმნის მეორე მეთოდია ფიგურული ფრჩხილების გამოყენება:

```
let user = {};
```

დღეისათვის უფრო გავრცელებულია მეორე ხერხი.

ობიექტის თვისებები

ობიექტის შექმნის შემდეგ შეგვიძლია განვსაზღვროთ მისი თვისებები. ამისათვის ობიექტის სახელის შემდეგ უნდა დავსვათ წერტილი, მივუთითოთ თვისების სახელი და მივანიჭოთ მნიშვნელობა:

```
let user = {};
user.name = "Tom";
user.age = 26;
```

მოცემულ შემთხვევაში გამოცხადებულია ორი თვისება name და age, რომელთაც მინიჭებული აქვთ შესაბამისი მნიშვნელობები. ამის შემდეგ ჩვენ შეგვიძლია გამოვიყენოთ ეს თვისებები, მაგალითად გამოვიტანოთ კონსოლში:

```
console.log(user.name);
```

```
console.log(user.age);
```

ობიექტის მეთოდები

ობიექტის მეთოდები განსაზღვრავენ მის ქცევას ან მოქმედებას. მეთოდები თავისი არსით წარმოადგენენ ფუნქციებს. მაგალითად, განვსაზღვროთ მეთოდი, რომელიც კონსოლსი გამოიტანს ადამიანის სახელს და ასაკს:

```
let user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){
  console.log(user.name);
  console.log(user.age);
};

// მეთოდის გამოძახება
user.display();
```

როგორც ფუნქციის შემთხვევაში, მეთოდი ჯერ განისაზღვრება, ხოლო შემდეგ ხდება მისი გამოძახება.

მასივის სინტაქსი

არსებობს თვისებების და მეთოდების განსაზღვრის ალტერნატიული გზაც მასივის სინტაქსის მეშვეობით:

```
let user = {};
user["name"] = "Tom";
user["age"] = 26;
user["display"] = function(){
  console.log(user.name);
  console.log(user.age);
};

// მეთოდის გამოძახება
user["display"]();
```

თითოეული თვისების და მეთოდის დასახელება ექცევა ბრჭყალებში და კვადრატულ ფრჩხილებში, შემდეგ ასევე ენიჭება მნიშვნელობა. მაგალითად: `user["age"] = 26`.

ამ თვისებებისა და მეთოდებისადმი მიმართვის უნდა გამოვიყენოთ წერტილის ნოტაცია `(user.name)` ან მივმართოთ ასე:`user["name"]`.

ობიექტის შექმნის გამარტივებული სინტაქსი

ობიექტის თვისებებისა და მეთოდების ცალ-ცალკე განსაზღვრის ნაცვლად შეგვიძლია ისინი განვსაზღვროთ ერთდროულად:

```
let user = {
  name: "Tom",
  age: 26,
  display: function () {
    console.log(this.name);
    console.log(this.age);
     }
};
// მეთოდის გამოძახება
user.display();
```

ობიექტის განსაზღვრის ასეთი მეთოდის დროს მინიჭებბის ოპერატორად გამოიყენება არა ტოლობის ნიშანი, არამედ ორწერტილი. მაგალითად, `name:"Tom"` გამოსახულება თვისება `name`-ს ანიჭებს მნიშვნელობას `"Tom"`. თითოეული თვისების ან მეთოდის განსაზღვრის შემდეგ მოდის არა წერტილმძიმე, არამედ მძიმე.

იმისათვის, რომ მივმართოთ ობიექტის თვისებას ან მეთოდს ამ ობიექტის შიგნით, გამოიყენება საკვაძო სიტყვა `this`. ის ნიშნავს მიმართვას მიმდინარე ობიექტისადმი.

აქვე უნდა აღინიშნოს, რომ ობიექტის თვისებებისა და მეთოდების სახელები წარმოადგენენ სტრიქონებს. ანუ ობიექტის წინა განსაზღვრა შეგვიძლია შევცვალოთ ასეთნაირად:

```
let user = {
  "name": "Tom",
  "age": 26,
  "display": function () {
    console.log(user.name);
    console.log(user.age);
     }
};
// მეთოდის გამოძახება
user.display();
```

ერთის მხრივ, ამ ორ განსაზღვრას შორის არანაირი განსხვავება არ არის. მეორეს მხრივ, არის შემთხვევები, როცა სახელების სტრიქონებში ჩასმამ შეიძლება გვიშველოს. მაგალითად იმ შემთხვევაში, როცა თვისების ან მეთოდის სახელი შედგება ცარიელი ადგილით გაყოფილი ორი სიტყვისგან:

```
let user = {
  name: "Tom",
```

```
  age: 26,
  "full name": "Tom Johns",
  "display info": function () {
    console.log(user.name);
    console.log(user.age);
     }
};
console.log(user["full name"]);
user["display info"]();
```

ოღონდ ასეთ შემთხვევაში თვისების ან მეთოდისადმი მიმართვისთვის უნდა გამოვიყენოთ მასივის სინტაქსი.

თვისებების და მეთოდების წაშლა

ზემოთ ჩვენ ვნახეთ, როგორ შეიძლება ობიექტს დინამიურად დავამატოთ მეთოდები და თვისებები. ასევე ჩვენ შეგვიძლია მათი წაშლაც. ამისთვის გამოიყენება ოპერატორი `delete`:

```
let user = {};
user.name = "Tom";
user.age = 26;
user.display = function () {
  console.log(user.name);
  console.log(user.age);
};

console.log(user.name); // Tom
delete user.name; // წავშალოთ თვისება
console.log(user.name); // undefined
```

წაშლის შემდეგ ობიექტის ეს თვისება არ იქნება განსაზღვრული და მასზე მიმართვისას დაბრუნდება `undefined`.

ჩასმული ობიექტები და მასივები ობიექტებში
ერთი ობიექტი თვისების სახით შეიძლება შეიცავდეს სხვა ობიექტს. მაგალითად, ავიღოთ ობიექტი `country` (ქვეყანა). მას შეიძლება ჰქონდეს სხვადასხვა თვისება. ერთ-ერთი თვისება შეიძლება იყოს დედაქალაქი. მაგრამ დედაქალაქს თავის მხრივ შეიძლება ჰქონდეს თვისებები, მაგალითად დასახელება, მოსახლეობის რაოდენობა, დაარსების წელი:

```
let country = {
  name: "გერმანია",
  language: "გერმანული",
  capital: {
    name: "ბერლინი",
```

```
    population: 3375000,
    year: 1237
     }
};
console.log("დედაქალაქი: " + country.capital.name); // Берлин
console.log("მოსახლეობა: " + country["capital"]["population"]); // 337500
console.log("დაარსების წელი: " + country.capital["year"]); // 1237
```

ჩასმული ობიექტის თვისებისადმი მიმართვისთვის შეგვიძლია გამოვიყენოთ სტანდარტული წერტილის ნოტაცია:

```
country.capital.name
```

ან მიმვმართოთ მასივის სინტაქსით:

```
country["capital"]["population"]
```

დასაშვებია მიმართვის შერეული სახეც:

```
country.capital["year"]
```

თვისებად ასევე შეიძლება მასივების გამოყენებაც, მათ შორის ობიექტების მასივებისაც:

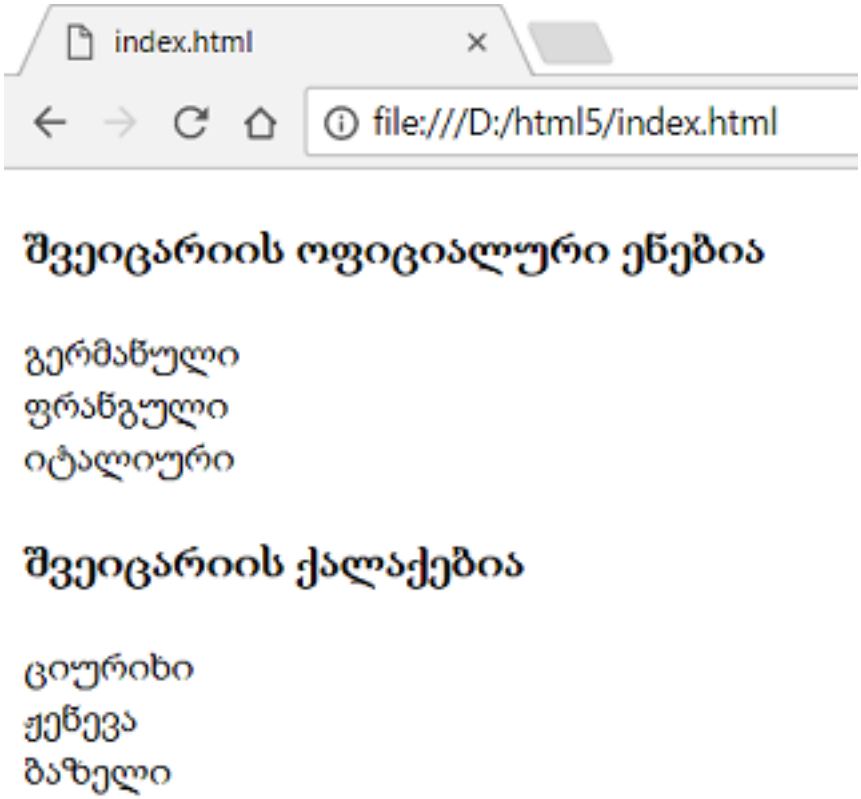
```
let country = {
  name: "შვეიცარია",
  languages: ["გერმანული", "ფრანგული", "იტალიური"],
  capital: {
    name: "ბერნი",
    population: 126598
     },
  cities: [
        { name: "ციურიხი", population: 378884 },
        { name: "ჟენევა", population: 188634 },
        { name: "ბაზელი", population: 164937 }
    ]
};

// country.languages ყველა ელემენტის გამოტანა
document.write("<h3>შვეიცარიის ოფიციალური ენებია</h3>");
for (let i = 0; i < country.languages.length; i++)
  document.write(country.languages[i] + "<br/>");

// country.cities ყველა ელემენტის გამოტანა
document.write("<h3>შვეიცარიის ქალაქებია</h3>");
for (let i = 0; i < country.cities.length; i++)
```

```
    document.write(country.cities[i].name + "<br/>");
```

`country` ობიექტში გვაქვს თვისება `languages`, რომელიც შეიცავს სტრიქონების მასივს, ასევე თვისება `cities`, რომელიც შეიცავს ერთი ტიპის ობიექტების მასივს. ამ მასივებთან შეგვიძლია ვიმუშაოთ ისევე, როგორც სხვა მსივებთან, მაგალითად გადავარჩიოთ `for` ციკლის მეშვეობით. მასივის ობიექტების გადარჩევისას თითოეული ელემენტი წარმოადგენს ობიექტს და შეგვიძლია მივმართოთ მის თვისებებს:

```
country.cities[i].name
```

შედეგად ბრაუზერი გამოიტანს ამ მასივების შიგთავსს:



მეთოდებისა და თვისებების არსებობის შემოწმება და გადარჩევა
ობიექტში ახალი თვისებებისა და მეთოდების დინამიურად დამატების დროს მნიშვნელოვანია შემოწმდეს, ხომ არა აქვს ასეთი თვისება ან მეთოდი ობიექტს უკვე. ამისათვის ჯავასკრიპტში გამოიყენება ოპერატორი `in`:

```
let user = {};
user.name = "Tom";
user.age = 26;
user.display = function () {
  console.log(user.name);
```

```
  console.log(user.age);
};
let hasNameProp = "name" in user;
console.log(hasNameProp); // true - თვისება name უკვე არსებობს user ობიექტ
let hasWeightProp = "weight" in user;
console.log(hasWeightProp); // false - user ობიექტში არაა თვისება ან მეთოდი
```

in ოპერატორს გააჩნია შემდეგი სინტაქსი: "თვისება|მეთოდი" in ობიექტი – ბრჭყალებში მოდის მეთოდის ან თვისების სახელი, ხოლო in ოპერატორის მერე ობიექტის სახელი. თუ თვისება ან მეთოდი ამ სახელით მოიძებნება ობიექტში, ბრუნდება true, წინააღმდეგ შემთხვევაში – false.

ალტერნატიული ხერხი მდგომარეობს შემდეგში: თუ ობიექტის თვისება ან მეთოდი არის undefined, ესე იგი, ეს თვისება ან მეთოდი ობიექტს არ გააჩნია:

```
let hasNameProp = user.name!==undefined;
console.log(hasNameProp); // true
let hasWeightProp = user.weight!==undefined;
console.log(hasWeightProp); // false
```

რადგანაც ობიექტს გააჩნია ტიპი Object, მას ასევე აქვს Object ტიპის ყველა თვისება და მეთოდი. შეგვიძლია გამოვიყენოთ მეტოდი hasOwnProperty(), რომელიც განსაზღვრულია Object ტიპში:

```
var hasNameProp = user.hasOwnProperty('name');
console.log(hasNameProp); // true
var hasDisplayProp = user.hasOwnProperty('display');
console.log(hasDisplayProp); // true
var hasWeightProp = user.hasOwnProperty('weight');
console.log(hasWeightProp); // false
```

თვისებებისა და მეთოდების გადარჩევა

for ციკლის მეშვეობით შესაძლებელია ობიექტის გადარჩევა, როგორც ჩვეულებრივი მასივისა და მისი ყველა თვისებისა და მეთოდის მიღება მათი მნიშვნელობებით:

```
var user = {};
user.name = "Tom";
user.age = 26;
user.display = function () {
  console.log(user.name);
  console.log(user.age);
};
for (var key in user) {
```

```
    console.log(key + " : " + user[key]);
}
```

გაშვებისას ბრაუზერის კონსოლი გამოიტანს შემდეგს:

```
name : Tom
age : 26
display : function (){

      console.log(user.name);
      console.log(user.age);
```

ობიექტები ფუნქციებში

ფუნქცია შეიძლება აბრუნებდეს მნიშვნელობას. არაა აუცილებელი, რომ ეს იყოს პრიმიტიული მონაცემები – რიცხვები, სტრიქონები, ასევე შეიძლება იყოს რთული ობიექტები.

მაგალითად, გავიტანოთ `User` ობიექტის შექმნა ფუნქციაში:

```
function createUser(pName, pAge) {
  return {
    name: pName,
    age: pAge,
    displayInfo: function() {
      document.write("სახელი: " + this.name + " ასაკი: " + this.age + "<br/
        }
     };
};
var tom = createUser("Tom", 26);
tom.displayInfo();
var alice = createUser("Alice", 24);
alice.displayInfo();
```

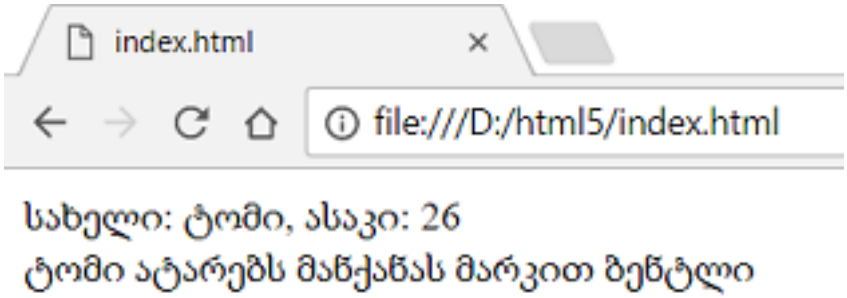
აქ ფუნქცია `createUser()` პარამეტრებად იღებს `pName`-ს და `pAge`-ს და მათ მიხედვით ქმნის ახალ ობიექტს, რაც წარმოადგენს დასაბრუნებელ მნიშვნელობას. ობიექტის შექმნის პროცესის ფუნქციაში გატანას ის უპირატესობა აქვს, რომ მისი მეშვეობით შესაძლებელია ერთი ტიპის ბევრი ობიექტის შექმნა სხვადასხვა მნიშვნელობებით.

ასევე, ობიექტი შეიძლება იყოს პარამეტრად ფუნქციაში:

```
function createUser(pName, pAge) {
  return {
    name: pName,
    age: pAge,
    displayInfo: function () {
      document.write("სახელი: " + this.name + ", ასაკი: " + this.age + "<br
```

```
        },
    driveCar: function (car) {
      document.write(this.name + " ატარებს მანქანას მარკით " + car.name + "<b
        }
    };
};

function createCar(mName, mYear) {
  return {
    name: mName,
    year: mYear
    };
};
var tom = createUser("ტომი", 26);
tom.displayInfo();
var bently = createCar("ბენტლი", 2004);
tom.driveCar(bently);
```

აქ განსაზღვრულია ორი ფუნქცია: ერთი მომხმარებელთა დასამატებლად, მეორე მანქანის ობიექტის შესაქმნელად. `user` ობიექტის `driveCar()` მეთოდს პარამეტრად გადაეცემა ობიექტი `car`. შედეგად, ბრაუზერსი გამოვა შემდეგი:



ობიექტის კონსტრუქტორი

ახალი ობიექტების შექმნის გარდა JavaScript საშუალებას გვაძლევს შევქმნათ ობიექტის ახალი ტიპები კონსტრუქტორის მეშვეობით. ობიექტის შექმნის ერთ-ერთი ხერხია მისი შექმნა `Object`-ის კონსტრუქტორის მეშვეობით:

```
var tom = new Object();
```

შექმნის შემდეგ ის იქცევა, როგორც `Object` ტიპის ობიექტი.

კონსტრუქტორი გვაძლევს ახალი ტიპის ობიექტის შექმნის შესაძლებლობას. ტიპი წარმოადგენს აბსტრაქტულ აღწერას ანუ ობიექტის შაბლონს. შეიძლება ასეთი ანალოგიის მოყვანაც. ყველას გვაქვს წარმოდგენა ადამიანზე – მას

უნდა ჰქონდეს ორი ხელი, ორი ფეხი, თავი, ტანი, სისხლის მიმოქცევის სისტემა, ნერვული სისტემა და ა. შ. ეს ერის ერთგვარი შაბლონი – ეს შაბლონი შეიძლება განვიხილოთ როგორც ობიექტის ტიპი. რეალურად არსებული ადამიანი კი არის ამ ტიპის კონკრეტული ობიექტი.

ტიპის განსაზღვრა შეიძლება შედგებოდეს კონსტრუქტორის ფუნქციისგან, მეთოდებისა და თვისებებისგან.

პირველ რიგში განვსაზღვროთ კონსტრუქტორი:

```
function User(pName, pAge) {
  this.name = pName;
  this.age = pAge;
  this.displayInfo = function(){
    document.write("სახელი: " + this.name + "; ასაკი: " + this.age + "<br/>
     };
}
```

კონსტრუქტორი – ეს ჩვეულებრივი ფუნქციაა, იმ განსხვავებით, რომ მას აქვს თვისებები და მეთოდები. თვისების და მეთოდის განსაზღვრისთვის გამოიყენება საკვანძო სიტყვა `this`:

```
this.name = pName;
```

მოცემულ შემთხვევაში განსაზღვრულია ორი თვისება – `name` და `age`, და ერთი მეთოდი – `displayInfo`.

როგორც წესი, კონსტრუქტორის სახელი იწყება დიდი ასოთი, განსხვავებით ფუნქციის სახელებისგან. ამის შემდეგ ამ კონსტრუქტორის მეშვეობით შეგვიძლია შევქმნათ ახალი `User` ტიპის ობიექტები და გამოვიყენოთ მისი თვისებები და მეთოდები:

```
var tom = new User("Том", 26);
console.log(tom.name); // Том
tom.displayInfo();
```

კონსტრუქტორის გამოსაძახებლად, ანუ ახალი `User` ობიექტის შესაქმნელად უნდა გამოვიყენოთ საკვანძო სიტყვა `new`.

ანალოგიურად შეგვიძლია განვსაზღვროთ სხვა ტიპებიც და გამოვიყენოთ ისინი ერთად:

```
// Car ტიპის კონსტრუქტორი
function Car(mName, mYear){
  this.name = mName;
```

```
  this.year = mYear;
  this.getCarInfo = function(){
    document.write("მოდელი: " + this.name + "  გამოშვების წელი: " + this.y
     };
};
// User ტიპის კონსტრუქტორი
function User(pName, pAge) {
  this.name = pName;
  this.age = pAge;
  this.driveCar = function(car){
    document.write(this.name + " ატარებს მანქანას მარკით " + car.name + "<br/
     };
  this.displayInfo = function(){
    document.write("სახელი: " + this.name + "; ასაკი: " + this.age + "<br/>
     };
};

var tom = new User("ტომი", 26);
tom.displayInfo();
var bently = new Car("ბენტლი", 2004);
tom.driveCar(bently);
```

ობიექტების გაფართოება. `Prototype`
გარდა თვისებებისა და მეთოდების განსაზღვრისა, კონსტრუქტორში ასევე შეიძლება გამოვიყენოთ თვისება `prototype`. ყველა ფუნქციას გააჩნია თვისება `prototype`, რომელიც წარმოადგენს ფუნქციის პროტოტიპს. ანუ თვისება `User.prototype` წარმოადგენს `User` ობიექტების პროტოტიპს. ყველა თვისება და მეთოდი, რომელიც განსაზღვრულია `User.prototype`-ში, იქნება საერთო ყველა `User` ობიექტისთვის.

მაგალითად:

```
function User(pName, pAge) {
  this.name = pName;
  this.age = pAge;
  this.displayInfo = function(){
    document.write("სახელი: " + this.name + "; ასაკი: " + this.age + "<br/>
     };
};

User.prototype.hello = function(){
  document.write(this.name + " ამბობს: 'გამარჯობა!'<br/>");
};
User.prototype.maxAge = 110;

var tom = new User("ტომი", 26);
```

```
tom.hello();

var john = new Uინკაფსულაცია
```

ინკაფსულაცია წარმოადგენს ობიექტზე ორიენტირებული პროგრამირების ერთ-ერთ საკვან
ობიექტის მდგომარეობის დაფარვას გარე ზემოქმედებისაგან. მიუთითებლობისას ო
წარმოადგენს საჯაროს და მათზე მიმართვა შესაძლებელია პროგრამის ნებისმიერი ადგი

```
function User(pName, pAge) {
  this.name = pName;
  this.age = pAge;
  this.displayInfo = function(){
    document.write("სახელი: " + this.name + "; ასაკი: " + this.age);
     };
};
var tom = new User("ტომი", 26);
tom.name=34;
console.log(tom.name);
```

მაგრამ ჩვენ შეგვიძლია გავხადოთ ისინი გარედან მიუწვდომელნი თვისებების ლოკალურ
გზით:

```
function User (name, age) {
  this.name = name;
  var _age = age;
  this.displayInfo = function(){
    document.write("სახელი: " + this.name + "; ასაკი: " + _age + "<br>");
     };
  this.getAge = function() {
    return _age;
     }
  this.setAge = function(age) {
    if(typeof age === "number" && age >0 && age<110){
      return _age = age;
         } else {
      console.log("არასწორი მნიშვნელობა");
         }
     }
}

var tom = new User("Том", 26);
console.log(tom._age); // undefined - _age - ლოკალური ცვლადი
console.log(tom.getAge()); // 26
tom.setAge(32);
console.log(tom.getAge()); // 32
tom.setAge("54"); // არასწორი მნიშვნელობა
```

კონსტრუქტორში თვისების age ნაცვლად გამოცხადებულია ლოკალური ცვლადი _age. რ
ცვლადების სახელები კონსტრუქტორში იწყება "_" სიმბოლოთი.

კონსტრუქტორის გარედან მომხმარებლის ასაკთან სამუშაოდ კონსტრუქტორში განსაზ
მეთოდი getAge() განკუთვნილია ლოკალური ცვლადის _age მნიშვნელობის მისაღება
გეტერს (getter). მეორე მეთოდი - setAge რომელსაც სეტერს (setter) უწოდებ
ცვლადისთვის მნიშვნელობის მისანიჭებლად;

ასეტი მიდგომის პლუსი მდგომარეობს იმაში, რომ ჩვენ შეგვიძლია ვაკონტროლოთ _
მინიჭების პროცესი, ჩვენ შეგვიძლია შევამოწმოთ მისანიჭებელი მნიშვნელობები, როგორ
სადაც მოწმდება, რომ ასაკი უნდა იყოს 0-ზე მეტი და 110-ზე ნაკლები რიცხვი.

```
ser("ჯონი", 28);
john.hello();
console.log(tom.maxAge); // 110
console.log(john.maxAge); // 110
```

აქ დამატებულია მეთოდი hello და თვისება maxAge და ნებისმიერი ობიექტი User იყენებს მათ. მნიშვნელოვანია აღვნიშნოთ, რომ ეს თვისება ერთი და იგივე იქნება ყველა User ობიექტისთვის, იგი არის სტატიკური თვისება, განსხვავებით სხვა თვისებებისგან, მაგალითად this.name, რომელიც ყველა ობიექტს თავისი აქვს.

ამავდროულად, ჩვენ შეგვიძლია თვითონ ობიექტში განვსაზღვროთ თვისება, რომელსაც ექნება იგივე სახელი, რაც პროტოტიპის თვისებას. ამ შემთხვევაში ობიექტის საკუთარ თვისებას ექნება პრიორიტეტი პროტოტიპის თვისებასთან შედარებით:

```
User.prototype.maxAge = 110;
var tom = new User("ტომი", 26);
var john = new User("ჯონი", 28);
tom.maxAge = 99;
console.log(tom.maxAge); // 99
console.log(john.maxAge); // 110
```

maxAge თვისებასთან მიმართვის დროს ჯავასკრიპტი ჯერ ეძებს თვიტონ ობიექტის თვისებებში და თუ ვერ იპოვის, მხოლოდ შემდეგ ეძებს პროტოტიპის თვისებებში. იგივე ეხება მეთოდებსაც.

ფუნქცია, როგორც ობიექტი. მეთოდები call და apply
ჯავასკრიპტში ფუნქციებიც წარმოადგენენ ობიექტებს - ობექტებს Function. მათ ასევე აქვთ პროტოტიპი, თვისებები და მეთოდები. ყველა ფუნქცია,

რომელიც გამოიყენება პროგრამაში, წარმოადგენს Function-ის ობიექტებს და აქვთ მისი ყველა თვისება და მეთოდი.

მაგალითად, ჩვენ შეგვიძლია ფუნქციის შექმნა Function-ის კონსტრუქტორის მეშვეობით:

```
var square = new Function('n', 'return n * n;');
console.log(square(5));
```

Function-ის კონსტრუქტორში შეიძლება გადაეცეს რამდენიმე პარამეტრი.ბოლო პარამეტრი წარმოადგენს ფუნქციის ტანს სტრიქონის სახით. ფაქტიურად, ეს სტრიქონი შეიცავს ჯავასკრიპტის კოდს. წინა არგუმენტები შეიცავენ ფუნქციის პარამეტრების სახელებს. მოცემულ მაგალითში განსაზღვრულია რიცხვის კვადრატში აყვანის ფუნქცია და მას გააჩნია ერთი პარამეტრი n.

Function ობიექტის თვისებებს შორის შეიძლება გამოვყოთ:

- arguments: პარამეტრების მასივი
- length: პარამეტრების რაოდენობა, რასაც ელოდება ფუნქცია
- caller: განსაზღვრავს ფუნქციას, რომელმაც წამოიწყო ფუნქციის მიმდინარე შესრულება
- name: ფუნქციის სახელი
- prototype: ფუნქციის პროტოტიპი

პროტოტიპის მეშვეობით ჩვენ შეგვიძლია განვსაზღვროთ დამატებითი თვისებები:

```
function display(){
  console.log("გამარჯობა, სამყაროვ!");
}
Function.prototype.program ="Hello";

console.log(display.program); // Hello
```

მეთოდებს შორის აღსანიშნავია call() და apply().

მეთოდი call() იძახებს ფუნქციას მითითებული მნიშვნელობით this და პარამეტრებით:

```
function add(x, y){
  return x + y;
}
var result = add.call(this, 3, 8);

```

```
console.log(result); // 11
```

this უთითებს ობიექტზე, რომლისთვისაც ხდება ფუნქციის გამოძახება – მოცემულ შემთხვევაში გლობალური ობიექტი window. შემდეგ მოდის პარამეტრების მნიშვნელობები.

პირველი პარამეტრის მეშვეობით ობიექტის გადაცემისას მასზე მიმართვა შეიძლება საკვანძო სიტყვის – this მეშვეობით:

```
function User (name, age) {
  this.name = name;
  this.age = age;
}
var tom = new User("ტომი", 26);
function display(){
  console.log("მისი სახელია " + this.name);
}
display.call(tom); // მისი სახელია ტომი
```

მოცემულ შემთხვევაში გადაეცემა მხოლოდ ერთი მნიშვნელობა, ვინაიდან ფუნქციას პარამეტრები არ გააჩნია. ანუ ფუნქციის გამოძახება ხდება ობიექტისთვის tom.

თუ არა აქვს მნიშვნელობა ობიექტს, გადაეცემა null:

```
function add(x, y){
  return x + y;
}
var result = add.call(null, 3, 8);

console.log(result); // 11
```

მეთოდი apply() მეთოდი call()-ის მსგავსია. აქაც პირველ პარამეტრად გადაეცემა ობიექტი, რომლისთვისაც ხდება ფუნქციის გამოძახება. ხოლო მეორე პარამეტრად გადაეცემა არგუმენტების მასივი:

```
function add(x, y){
  return x + y;
}
var result = add.apply(null, [3, 8]);

console.log(result); // 11
```

მემკვიდრეობითობა

JavaScript-ს გააჩნია მემკვიდრეობითობა, რაც საშუალებას იძლევა ობიექტების ახალი ტიპების შექმნისას, საჭიროებისას, შევინარჩუნოთ არსებული ტიპებისგან მემკვიდრეობით მიღებული ფუნქციონალი. მაგალითად, ვთქვათ გვაქვს ობიექტი User , რომელიც წარმოადგენსმომხმარებელს. ვთქვათ, ასევე გვაქვს ობიექტი Employee, რომელიც წარმოადგენს მომუშავეს. მაგრამ მომუშავე ამავე დროს მომხმარებელიცაა, შესაბამისად, მას უნდა ჰქონდეს ასევე User ობიექტის ყველა თვისება და მეთოდი. მაგალითად:

```
// მომხმარებლის კონსტრუქტორი
function User (name, age) {
  this.name = name;
  this.age = age;
  this.go = function(){document.write(this.name + " მიდის <br/>");}
  this.displayInfo = function(){
    document.write("სახელი: " + this.name + "; ასაკი: " + this.age + "<br/>
     };
}
User.prototype.maxage = 110;

// მომუშავის კონსტრუქტორი
function Employee(name, age, comp){
  User.call(this, name, age);
  this.company = comp;
  this.displayInfo = function(){
    document.write("სახელი:          " + this.name + "; ასაკი:          " +
" + this.company + "<br/>");
     };
}
Employee.prototype = Object.create(User.prototype);

var tom = new User("ტომი", 26);
var bill = new Employee("ბილი", 32, "Google");
tom.go();
bill.go();
tom.displayInfo();
bill.displayInfo();
console.log(bill.maxage);
```

აქ დასაწყისში განსაზღვრულია ტიპი User და მის პროტოტიპში დამატებულია თვისება maxage. შემდეგ მოდის Employee ტიპის განსაზღვრა. მის კონსტრუქტორში ხდება User-ის კონსტრუქტორის გამოძახება:

```
User.call(this, name, age);
```

პირველი პარამეტრის (this) გადაცემა შესაძლებლობას გვაძლევს გამოვიძახოთ User-ის კონსტრუქტორი შესაქმნელი Employee ობიექტისთვის. ამის შედეგად ყველა მეთოდი და თვისება, რაც გააჩნია User-ს ასევე გადადის Employee ობიექტზე.

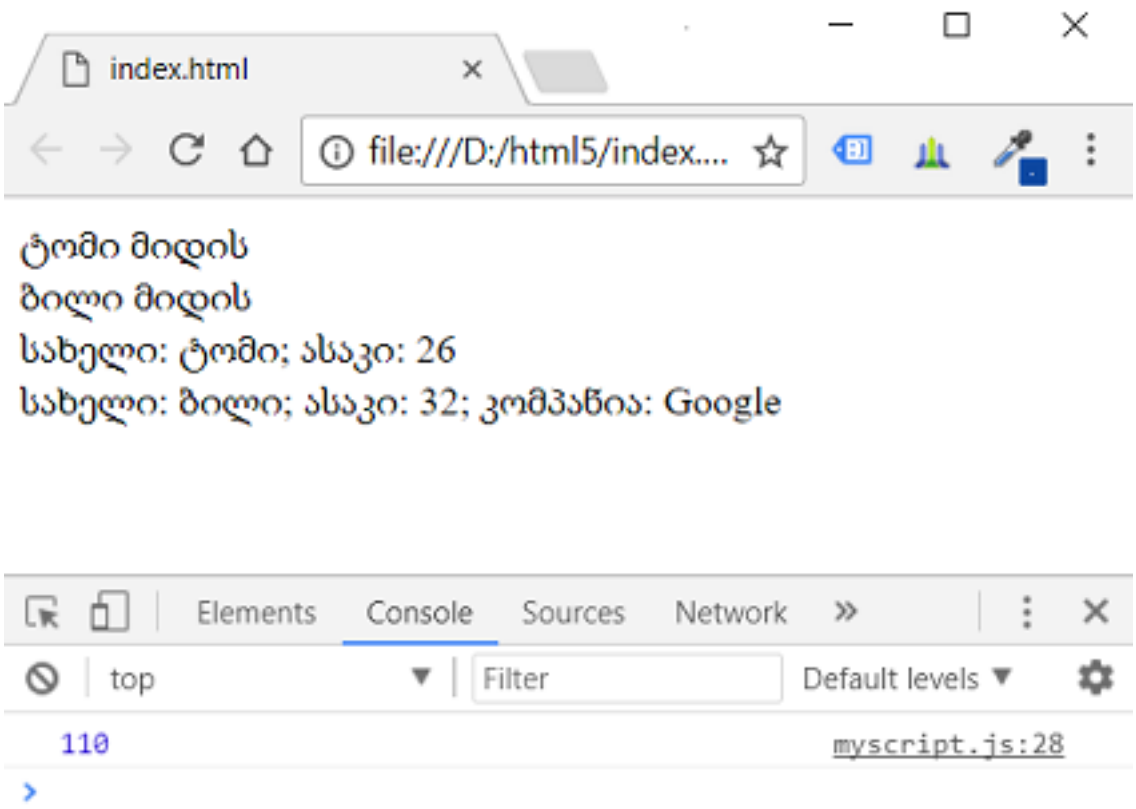
გარდა ამისა საჭიროა პროტოტიპის გადაცემა მემკვიდრეობით. ამისთვის გამოიყენება შემდეგი სტრიქონი:

```
Employee.prototype = Object.create(User.prototype);
```

მეთოდი Object.create() საშუალებას გვაძლევს შევქმნათ User-ის პროტოტიპის ობიექტი, რომელიც ენიჭება Employee-ს პროტოტიპს. ამასთან, Employee-ს პროტოტიპში, აუცილებლობისას, ჩვენ შეგვიძლია დამატებითი თვისებებისა და მეთოდების შექმნაც.

მემკვიდრეობითობისას შესაძლებელია მემკვიდრეობით მიღებული ფუნქციონალის ხელახალი განსაზღვრაც (შეცვლა). მაგალითად, Employee ობიექტში შეცვლილია მეთოდი displayinfo() და მასში დამატებულია მომუშავის კომპანიის გამოტანა.

ამ ყველაფრის შედეგად ბრაუზერში გამოვა შემდეგი ინფორმაცია:



საკვანძო სიტყვა this

this საკვანძი სიტყვის მნიშვნელობა დამოკიდებულია იმ კონტექსტზე, რომელშიც ხდება მისი გამოყენება, ასევე კოდის რეჟიმზე (strict mode) - მკაცრია იგი თუ არა.

გლობალური კონტექსტი

გლობალურ კონტექსტში this მიმართავს გლობალურ ობიექტს. ასეთ შემთხვევაში არა აქვს მნიშვნელობა, თუ რომელ რეჟიმში (მკაცრში თუ არამკაცრში) ხდება მისი გამოყენება.

```
this.alert("global alert");
this.console.log("global console");

var currentDocument = this.document;
```

მოცემულ შემთხვევაში this საკვანძო სიტყვის გამოყენება პრაქტიკულად ზედმეტია.

ფუნქციის კონტექსტი

თუ სკრიპტის გაშვება ხდება მკაცრ რეჟიმში (დირეტივა "use strict"), this მიმართავს უშუალოდ ფუნქციის კონტექსტს, წინააღმდეგ შემთხვევაში - გარე კონტექსტს. მაგალითად:

```
function foo(){
  var bar = "bar2";
  console.log(this.bar);
}

var bar = "bar1";

foo();  // bar1
```

მკაცრი რეჟიმის შემთხვევაში მივიღებდით შეცდომას undefined:

```
"use strict";
function foo(){
  var bar = "bar2";
  console.log(this.bar);
}

var bar = "bar1";

foo();  // შეცდომა - this - undefined
```

ობიექტის კონტექსტი

ობიექტის კონტექსტში, მათ შორის მეთოდებშიც, საკვანძო სიტყვა `this` მიმართავს თვითონ ამ ობიექტს:

```
var o = {
  bar: "bar3",
  foo: function(){
    console.log(this.bar);
     }
}
var bar = "bar1";
o.foo();    // bar3
```

მაგალითები

განვიხილოთ ცოტა უფრო რთული მაგალითი:

```
function foo(){
  var bar = "bar2";
  console.log(this.bar);
}

var o3 = {bar:"bar3", foo: foo};
var o4 = {bar:"bar4", foo: foo};

var bar = "bar1";

foo();      // bar1
o3.foo();   // bar3
o4.foo();   // bar4
```

აქ განსაზღვრულია გლობალური ცვლადი `bar`. ასევე, ფუნქციაში `foo()` გამოცხადებულია ლოკალური ცვლადი `bar`. რომელი ცვლადის მნიშვნელობას გამოიტანს ფუნქცია `foo`? ფუნქცია გამოიტანს გლობალურ ცვლადს, ვინაიდან სკრიპტი გაშვებულია რამკაცრ რეჟიმში, შესაბამისად საკვანძო სიტყვა `this` მიუთითებს გარე კონტექსტზე.

სხვანაირადაა საქმე ობიექტებთან მიმართებაში. თითოეულ ობიექტში განსაზღვრულია თავისი თვისება `bar`. `foo` მეთოდის გამოძახებისას ის მიმართავს ფუნქციის გარე კონტექსტს, რაც ამ შემთხვევაში არის შესაბამისი ობიექტის კონტექსტი.

ასეთმა ქცევამ ხანდახან შეიძლება გარკვეულ გაუგებრობამდე მიგვიყვანოს. განვიხილოთ ასეთი სიტუაცია:

```
var o1 = {
  bar: "bar1",
  foo: function () {
    console.log(this.bar);
     }
}
var o2 = { bar: "bar2", foo: o1.foo };

var bar = "bar3";
var foo = o1.foo;

o1.foo();   // bar1
o2.foo();   // bar2
foo();      // bar3
```

მიუხედავად იმისა, რომ ობიექტი `o2` იყენებს `foo` მეთოდს `o1` ობიექტიდან, `o1.foo` ფუნქცია `this` მნიშვნელობის ძებნას იწყებს გარე კონტექსტში, ანუ `o2` ობიექტის კონტექსტში. შესაბამისად ის აბრუნებს მნიშვნელობას `"bar2"`.

იგივე ეხება გლობალურ ცვლადს `foo`. `this.bar` მნიშვნელობის ძებნას იწყებს გარე კონტექსტში, ანუ გლობალურ კონტექსტში და შესაბამისად მნიშვნელობად იღებს `"bar3"`-ს.

თუმცა, თუ გამოვიძახებთ ფუნქციას მეორე ფუნქციიდან, გამოძახებული ფუნქცია ასევე მიმართავს გარე კონტექსტს:

```
var bar = "bar2";

function daz(){
  var bar = "bar5";
  function maz(){
    console.log(this.bar);
    }
  maz();
}
daz();  // bar2
```

აქ ფუნქცია `maz`-სთვის `this.bar` მიმართავს გარე კონტექსტს, ანუ `daz` ფუნქციის `this.bar`, თავის მხრივ, `daz` ფუნქციისთვის `this` მიმართავს გარე კონტექსტს, ანუ გლობალურ კონტექსტს, შესაბამისად საბოლოოდ ვღებულობთ `"bar2"` -ს და არა `"bar5"`-ს.

ცხადი მიბმა

`call()` და `apply()` მეთოდების მეშვეობით შეიძლება ფუნქცია მივაბათ განსაზღვრულ კონტექსტს:

```
function foo(){
  console.log(this.bar);
}

var o3 = {bar: "bar3"}
var bar = "bar1";
foo();  // bar1
foo.apply(o3);  // bar3
// ან
// foo.call(o3);
```

მეორე შემთხვევაში ფუნქცია `foo` მიბმულია `o3` ობიექტზე. ამიტომაც, ამ შემთხვევაში კონსოლში გამოიტანს `"bar3"`-ს.

მეთოდი bind

მეთოდი `f.bind(o)` საშუალებას გვაძლევს შევქმნათ ახალი ფუნქცია იგივე შიგთავსით და ხედვის იგივე არეალით, რაც აქვს `f` ფუნქციას, ოღონდ `o` ობიექტზე მიბმით:

```
function foo(){
  console.log(this.bar);
}

var o3 = {bar: "bar3"}
var bar = "bar1";
foo();  // bar1
var func = foo.bind(o3);
func(); // bar3
```