



Școala
informală
de IT

JS Objects and Classes

Agenda

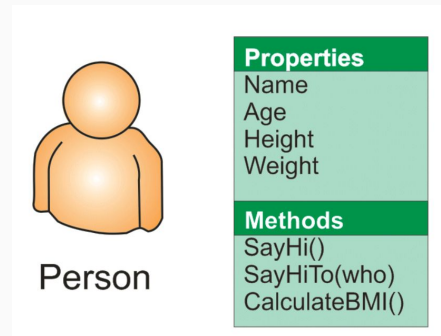
- **Objects**
- **Classes**
- **Prototypes**

Objects



Objects

- **An object (model) is the representation of a “real-life” object like: a car, a user, a dog, a flower.**
- **Objects in Javascript group values and/or functionality in order to build more complex data structures.**
- **Properties = data (variables)**
- **Methods = functionality (functions)**

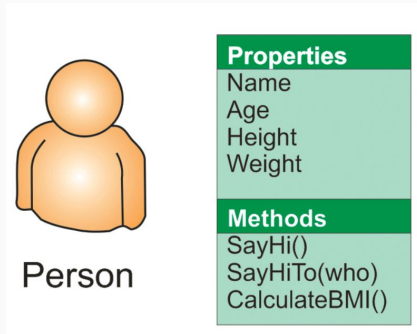


An object is the representation of a “real-life” object, like: a car, a user, a dog, a flower. From a Javascript standpoint, objects are used to group values and/or functionality in order to build more complex data structures.

Data storing is done via properties (a property is a variable inside an object), and the functionality is described using methods (a method is a function inside an object).

A person object could have name, age, height, weight as properties, and sayHi(), sayHiTo(who) and calculateBMI() would be its methods.


A Person in Javascript



```
var user = {  
  name: "John Doe",  
  age: 25,  
  height: 170,  
  weight: 70,  
  sayHi: function(){  
    console.log("Hi");  
  },  
  sayHiTo: function(name){  
    console.log("Hi " + name);  
  },  
  calculateBMI: function(){  
    return this.height / this.weight;  
  }  
}
```

The representation of Person's object in Javascript.

A Laptop in Javascript

Laptop	Properties	Method
	laptop.color = black	laptop.start()
	laptop.brand = Dell	laptop.shutdown()
	laptop.hardDiskSize = 200 GB	laptop.sleep()
	laptop.memory = 4 GB	laptop.playMusic()
	laptop.price = \$ 400	laptop.playVideo()

```
var laptop = {  
  color: "black",  
  brand: "Dell",  
  hardDiskSize: "200GB",  
  memory: "4GB",  
  price: 400,  
  start: function(){  
    console.log("starting laptop...");  
  },  
  ...  
}
```

```
...  
shutdown: function(){  
  console.log("shutting down...");  
},  
sleep: function(){  
  console.log("going to sleep...");  
},  
...
```

```
...  
playMusic: function(){  
  console.log("playing music, are you  
              ready to dance?");  
},  
playVideo: function(){  
  console.log("get your popcorn, i'm  
              playing some video here!");  
}  
}
```

The representation of Laptop's object in Javascript.

Accessing Object Properties

```
var user = {  
  name: "John Doe",  
  age: 25,  
  height: 170,  
  weight: 70,  
  sayHi: function(){  
    console.log("Hi");  
  },  
  sayHiTo: function(name){  
    console.log("Hi " + name);  
  },  
  calculateBMI: function(){  
    return this.height / this.weight;  
  }  
}
```

```
// Static (dot notation)  
user.name    // "John Doe"  
user.age     // 25  
user.sayHi() // Hi  
  
// same as (brackets notation)  
user["name"] // "John Doe"  
user["age"]  // 25  
user["sayHi"]() // Hi  
  
// Dynamic  
var prop = "name"; // prop value can change  
user[prop] // "John Doe"  
prop = "age";  
user[prop] // 25  
prop = "sayHi";  
user[prop]() // Hi
```

There are two ways of accessing object properties and methods: static and dynamic.

To get the value of a property or to call a method in a static way, you can either use the dot (.) notation or the brackets ([]) notation - similar to how the elements of an array are accessed, but instead of using an index, use the name of the property or method that you want to access or call.

The same brackets notation can be used to dynamically access object properties and methods: instead of using the strings with the name of the properties, you can use variables (whose values can be changed).

Object Built-In Methods

- **Object.**[keys\(\)](#)
- **Object.**[values\(\)](#)
- **Object.**[entries\(\)](#)

```
var user = {  
  name: "John Doe",  
  age: 25,  
  height: 170,  
  weight: 70,  
  sayHi: function(){...},  
  sayHiTo: function(name){...},  
  calculateBMI:function(){...}  
}
```

```
Object.keys(user)    // ["name", "age", "height", "weight", "sayHi", "sayHiTo", "calculateBMI"]  
Object.values(user)  // ["John Doe", 25, 170, 70, function(){...}, function(){...}, function(){...}]  
Object.entries(user) // [[ "name", "John Doe"], [ "age", 25], [ "height", 170], [ "weight", 70],  
                        [ "sayHi", function(){...}], [ "sayHiTo", function(){...}], [ "calculateBMI",  
                        function(){...}]]
```

There are several built-in methods that are available to be called on all objects. Three of the most important are:

- **Object.keys()** - returns an array of a given object's property **names**, in the same order as we get with a normal loop
- **Object.values()** - returns an array of a given object's own enumerable property **values**
- **Object.entries()** - returns an array of a given object's own enumerable property **[key, value]** pairs

These arrays are regular arrays, so all known array methods can be applied on them too.

Shortcomings

- What if we need multiple objects of the same type?
 - Code Repetition
 - Maintenance hell

Let's get back to users for our website example. I don't need all my users to have the same attributes (name,age,weight...) , but I do want all of them to be able to "sayHello", to have the possibility to calculate his BMI and so on...

```
var user1 = {  
  name: "John Doe",  
  age: 25,  
  height: 170,  
  weight: 70,  
  sayHi: function(){ console.log("Hi"); },  
  sayHiTo: function(name){ console.log("Hi "+name); },  
  calculateBMI: function(){ return this.height/this.weight; }  
}
```

```
var user2 = {  
  name: "Mary Anne",  
  age: 35,  
  height: 160,  
  weight: 50,  
  sayHi: function(){ console.log("Hi"); },  
  sayHiTo: function(name){ console.log("Hi "+name); },  
  calculateBMI: function(){ return this.height/this.weight; }  
}
```

```
var user3 = {  
  name: "James Arthur",  
  age: 32,  
  height: 180,  
  weight: 90,
```

```
sayHi: function(){ console.log("Hi"); },  
sayHiTo: function(name){ console.log("Hi "+name); },  
calculateBMI: function(){ return this.height/this.weight; }  
}
```

But what if I want to create 100 users? Or to change the formula for BMI calculation and we have multiple users on our website?! I have to manually write code to create all the users, or in the second case, I have to update all the “user” objects “calculateBMI” method. I definitely don’t want that.

Creating several objects of the same type

```
// factory function
function createNewUser(options) {
  var user = {};
  user.name = options.name || "";
  user.age = options.age;
  user.height = options.height;
  user.weight = options.weight;
  user.sayHi = function(){
    console.log("Hi");
  }
  return user;
}

var user1 = createNewUser({
  name: "John Doe",
  age: 25,
  height: 170,
  weight: 70
});
user1.name => "John Doe"
```

```
// constructor function
function User(options) {
  this.name = options.name || "";
  this.age = options.age;
  this.height = options.height;
  this.weight = options.weight;
  this.sayHi = function(){
    console.log("Hi");
  }
}

var user1 = new User({
  name: "John Doe",
  age: 25,
  height: 170,
  weight: 70
});
user1.name => "John Doe"
```

To create multiple users - all of them having the same properties and behavior, we create a function specialized in creating these types of objects: `createNewUser`. For this, we need to create an empty object, we add properties to it, and afterwards return that object. This is a bit too much to do.

JavaScript has a way to do this in a more straightforward way: using **constructor functions**. It has all the features you'd expect in a function, although it doesn't return anything or explicitly create an object — it basically just defines properties and methods. Its name usually starts with a capital letter — this convention is used to make constructor functions easier to recognize in code.

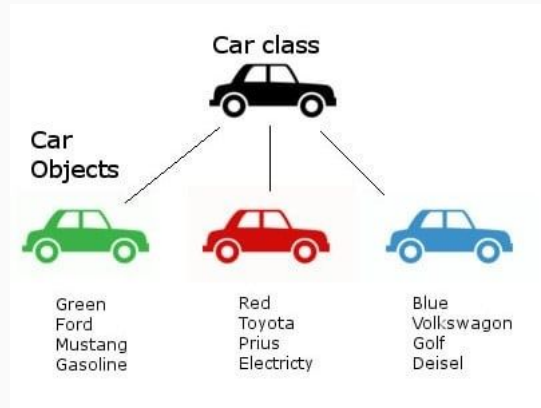
The constructor function is JavaScript's version of a **class**.

Classes



Class Definition

- A class is a blueprint or template or set of instructions to build a specific type of object.
- Every object is built from a class.
- (in other words,) An object is an instance of a class.



Classes

```
// constructor function
function User(options) {
  this.name = options.name || "";
  this.age = options.age;
  this.height = options.height;
  this.weight = options.weight;
  this.sayHi = function(){
    console.log("Hi");
  }
}
```

```
// add more class methods
User.prototype.sayHiTo = function(name){
  console.log("Hi " + name);
}
User.prototype.calculateBMI = function(){
  // intentionally incorrect expression
  return this.height / this.weight;
}
```

```
// instantiating an object
var john = new User({
  name: "John Doe",
  age: 25,
  height: 170,
  weight: 70
});

john.name => "John Doe"
john.calculateBMI() => 2.42
```

```
// instantiating an object
var mary = new User({
  name: "Mary Anne",
  age: 35,
  height: 160,
  weight: 50
});

mary.name => "Mary Anne"
mary.calculateBMI() => 3.2
```

```
// updating a method on all object instances
User.prototype.calculateBMI = function(){
  var heightInMeters = this.height/100;
  return this.weight/(heightInMeters * heightInMeters);
}

john.calculateBMI() => 24.22
mary.calculateBMI() => 19.53
```

Let's get back to our users. Looks like we need to define a **User** class and create users *instances* from it.

To define a User class, we'll use a **constructor function**. Inside it, we define the properties and the methods that each object of that type will have.

All properties and methods are set using the **this** keyword, which refers to the current object the code is being written inside — so in this case this is equivalent to **User**. **this** will always ensure that the correct values are used when a member's context changes (e.g. two different User object instances may have different heights and widths, but will want to use their own heights and widths when calculating their BMI).

User is just a template, we'll use it to create *actual objects*. This process is called **instantiation**. `user1` and `user2` are instances of **User**, created using the **new** keyword.

When we are calling our constructor function, we are defining `sayHi` every time, which isn't ideal. To *add* methods to **all** objects of the **User** type in an efficient way, we'll use **prototypes**.

We'll also use prototypes for *updating* methods for **all** objects (as in the `calculateBMI` example).

Prototype



Prototypes

- **Prototypes are the mechanism by which JavaScript objects inherit features from one another**
- **There are two interrelated concepts related to prototypes in JavaScript:**
 - **Prototype property**
 - **Prototype attribute**

Prototype Property

- Every JavaScript **function** has a **prototype property**
- You add **methods and properties** on a **function's prototype** to make those methods and properties available to **all** instances of that function

```
var f = function() {  
  console.log("check my prototype");  
}  
f.prototype => {} (Prototype is empty by default)  
  
function User(options) {  
  this.name = options.name || "";  
  this.age = options.age;  
  this.height = options.height;  
  this.weight = options.weight;  
  this.sayHi = function(){  
    console.log("Hi");  
  }  
}  
User.prototype => {constructor: f}  
                    constructor: f User(options)  
                    __proto__: Object
```

Every JavaScript **function** has a **prototype property**. It is empty by default, and you add methods and properties on a function's prototype property to make them available to all instances of that function.

The prototype has a constructor and a `__proto__` object, we'll discuss about them next.

Prototype Attribute

- Every **object** has a **prototype attribute**
- An object's prototype attribute points to the object's "**parent**" — the object it inherited its properties from

```
var userObj = new User({
  name: "John Doe",
  age: 25,
  height: 170,
  weight: 70
});

userObj.prototype => undefined
Object.getPrototypeOf(userObj) =>
  constructor: f User(options)
  __proto__: Object

// the same, but not supported in all browsers
userObj.__proto__

// both expressions below are true
User.prototype == Object.getPrototypeOf(userObj)
User.prototype.isPrototypeOf(userObj)
```

The second concept with prototypes in JavaScript is the **prototype attribute**. It is available on all objects, and it can be seen as a characteristic that points to the object's parent, i.e., the object it inherited its properties from.

The prototype attribute is usually referred to as the prototype object, and it's set automatically when you create a new object.

To access the prototype attribute, we need to use the **getPrototypeOf** method of the **Object** class. This returns the prototype of the constructor function that was used to create that object. **__proto__** can also be used (two underscores at the beginning and two at the end), it returns the same thing, but it's not supported in all browsers. This makes **getPrototypeOf** the recommended way of finding an object's prototype.

userObj is an instance of **User**. Thus, the prototype attribute of **userObj** is the same as the prototype property of the **User** function.

There is also a method that can be used to determine if a certain prototype property is the prototype attribute of a certain object: **isPrototypeOf**.

DOs and DON'Ts

- **Only modify your own prototypes.** Never modify the prototypes of standard JavaScript constructor functions like **Object**, **Array**, **Date**, **Function**.
- **Avoid assigning variables to prototype, use the constructor function instead.**

Of course, not only the `User` function has a prototype. We can see the prototypes of standard JavaScript constructor functions like `Object`, `Array`, `Date`, `Function`, using the same mechanisms:

`Object.prototype`

`Array.prototype`

`Date.prototype`

`Function.prototype`

It's important to remember to only modify the prototypes of the functions that you created. Never modify the prototypes of the standard JavaScript constructor functions.

Also, it's important to notice that assigning variables to a prototype is not necessarily very helpful. `User.prototype.name = "John Doe"` assigns the name `"John Doe"` to all instances of `User`. But the users will usually have different names, so this is not really a gain.

Instead, **use the constructor function to assign properties, and use the prototype attribute to assign methods to objects.**

`// constructor function`

```
function User(options) {
  this.name = options.name || "";
  this.age = options.age;
  this.height = options.height;
  this.weight = options.weight;
}

User.prototype.sayHi = function(){
  console.log("Hi");
}
User.prototype.sayHiTo = function(name){
  console.log("Hi " + name);
}
User.prototype.calculateBMI = function(){
  // intentionally incorrect expression
  return this.height / this.weight;
}
```

Constructors

- A constructor is a function used for initializing new objects
- You use the new keyword to call the constructor
- When calling `new User(...)`, 3 actions happen:
 - a. A new object is created
 - b. The prototype of that object is set to `User.prototype`
 - c. `User.prototype` is passed as `this` to the constructor

Prototype Chain

- When accessing an attribute or calling the method of an object, that is looked up on:
 - The object itself
 - If it does not exist there => on the prototype of the object
 - If it does not exist there => on the prototype of the prototype
 - ... and so on
- This is **the prototype chain**



If you want to access a property of an object, the search for the property begins directly on the object. If the JS runtime can't find the property there, it then looks for the property on the object's prototype - the object it inherited its properties from. If the property is not found on the object's prototype, the search for the property then moves to prototype of the object's prototype. And this continues until there is no more prototype.

This in essence is the **prototype chain**: the chain from an object's prototype to its prototype's prototype and onwards. And JavaScript uses this prototype chain to look for properties and methods of an object.

If the property does not exist on any of the object's prototype in its prototype chain, then the property does not exist and undefined is returned.

```
var car = {  
  color: "red",  
  brand: "VW"  
}
```

```
car.color // it's found directly on the object as we defined it
```

```
car.toString() // it's not found on the object itself so it goes up on the prototype  
chain to look for it; it will check on car prototype (creator). And since all  
objects created with the object literal inherit from Object.prototype, the toString  
method will be found on Object.prototype
```

```
car.maxSpeed // will return undefined as it's not defined on object and neither on  
it's parents
```

Recap



Recap

- An **object** has **properties** and **methods**.
- A **class** is a **blueprint** used to build a specific type of object.
- An object is an **instance** of a class.
- **Prototypes** are the mechanism by which objects inherit from one another.
- All functions have a prototype property (`User.prototype`)
- All objects have a prototype attribute (`Object.getPrototypeOf(userObj)`)
- When accessing an object property or method, it's searched at runtime on the object's **prototype chain**

Resources

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

<http://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/>