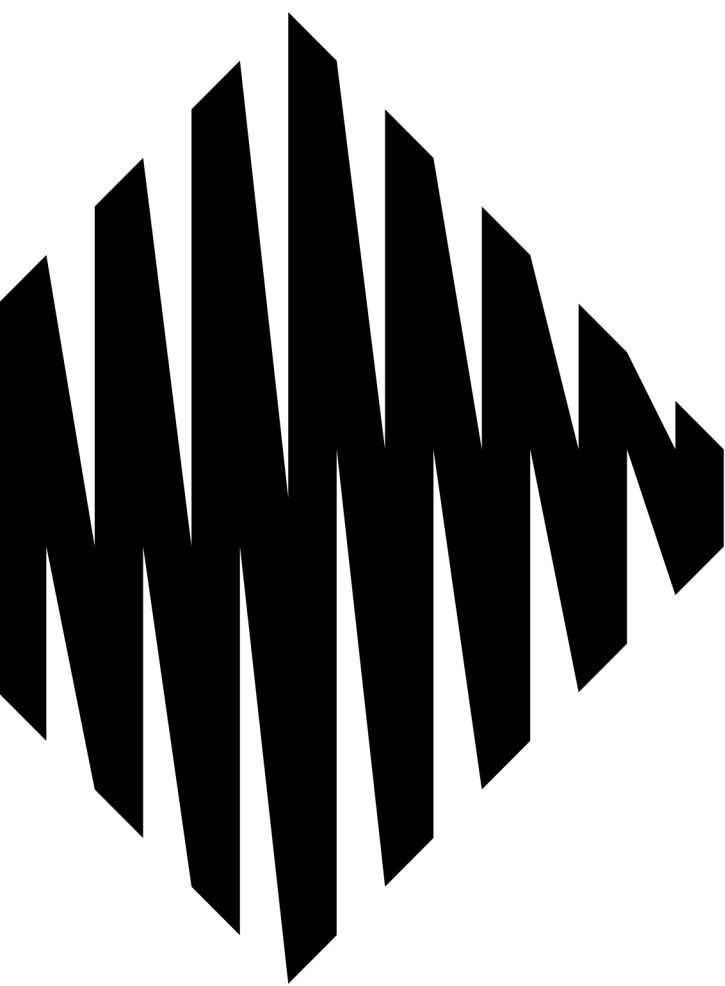


ecx.io Frontend Bootcamp

—
Javascript Day 05

21.07.2021.



Agenda

Objects

This

Immutability

ecx.io
an IBM Company

Objects

JavaScript object is a non-primitive data-type that allows you to store multiple collections of data

Objects are an unordered collection of properties, each of which has a name and a value

They are used to store various keyed collections and more complex entities.

Objects – how to create them?

Objects can be created in two ways:

Using object initializers

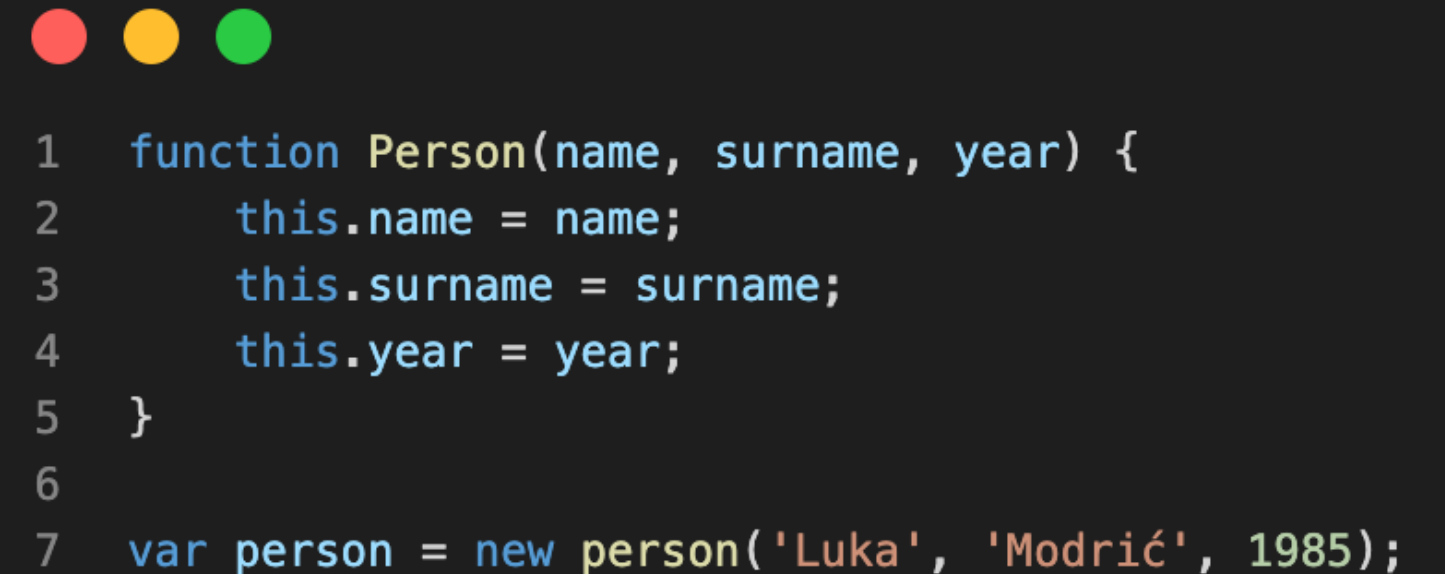
- Using object initializers is referred to creating objects with literal notation
- Objects made from object literal expressions are instances of Object
- Identical object initializers create distinct objects that will not compare to each other as equal

```
1  let person = {
2    name: 'Luka',
3    surname: 'Modrić',
4    year: 1985,
5    address: {
6      street: 'Ilica',
7      city: 'Zagreb'
8    },
9    getYears: function() {
10     return this.year;
11   }
12 }
```

Objects – how to create them?

Using a constructor function

- To define an object type, create a function for the object type that specifies its name, properties, and methods
- Notice the use of `this` to assign values to the object's properties based on the values passed to the function
- The construct function is called by the `new` operator



```
1  function Person(name, surname, year) {  
2      this.name = name;  
3      this.surname = surname;  
4      this.year = year;  
5  }  
6  
7  var person = new person('Luka', 'Modrić', 1985);
```

Factory functions



```
1  function person(firstName, lastName, age) {  
2    const person = {};  
3    person.firstName = firstName;  
4    person.lastName = lastName;  
5    person.age = age;  
6    return person;  
7  }
```

A **factory function** is any function which is not a class or constructor that returns a (presumably new) object. In JavaScript, any function can return an object. When it does so without the `new` keyword, it's a factory function.

If you look at the code on the left inside the function, it creates a new object and attached passed arguments to that object as properties to that and return the new object. That is a simple factory function in JavaScript.

Prototype

- JavaScript is often described as a **prototype-based language** — to provide inheritance, objects can have a **prototype object**, which acts as a template object that it inherits methods and properties from.
- An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on. This is often referred to as a **prototype chain**, and explains why different objects have properties and methods defined on other objects available to them.

Lets have a look at our example



```
1 function Animal(name, breed, kind, age) {  
2   this.name = name;  
3   this.breed = breed;  
4   this.kind = kind;  
5   this.age = age;  
6 }  
7  
8 const animal = new Animal('Dodo', 'doda-bird', 'bird', '125')  
9  
10 // try to type animal. and see what appears
```

If you type *animal* into your JavaScript console, you should see the browser try to auto-complete this with the member names available on this object:

```
> function Animal(name, breed, kind, age) {  
  this.name = name;  
  this.breed = breed;  
  this.kind = kind;  
  this.age = age;  
}  
  
const animal = new Animal('Snoopy', 'beagle', 'dog', '4')
```

< undefined

> animal.age

< "4"

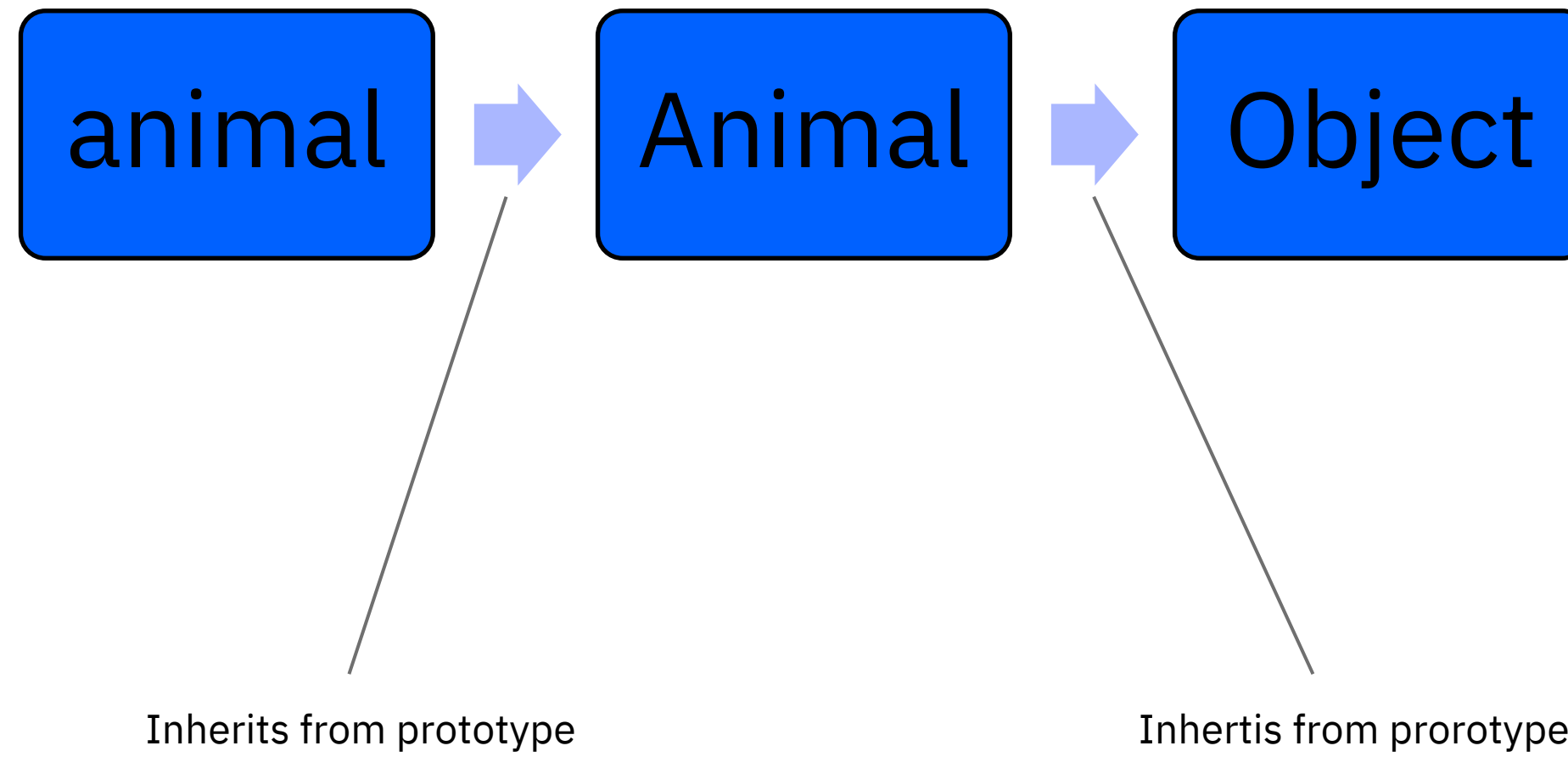
age	
breed	
kind	
name	
constructor	Animal
hasOwnProperty	Object
isPrototypeOf	
propertyIsEnumerable	
toLocaleString	
toString	
valueOf	
__defineGetter__	
__defineSetter__	
__lookupGetter__	
__lookupSetter__	
__proto__	


```
> function Animal(name, breed, kind, age) {  
  this.name = name;  
  this.breed = breed;  
  this.kind = kind;  
  this.age = age;  
}  
  
const animal = new Animal('Snoopy', 'beagle', 'dog', '4')  
< undefined  
> animal.age  
< "4"
```

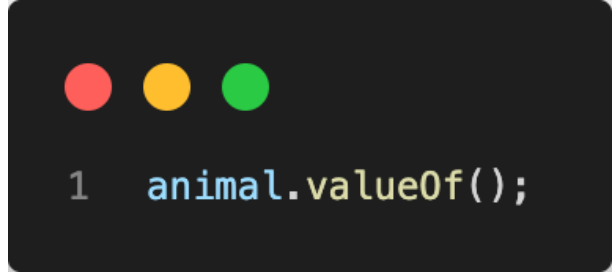
age	
breed	
kind	
name	
constructor	Animal
hasOwnProperty	Object
isPrototypeOf	
propertyIsEnumerable	
toLocaleString	
toString	
valueOf	
__defineGetter__	
__defineSetter__	
__lookupGetter__	
__lookupSetter__	
__proto__	

In this list, you will see the members defined on *animal* constructor — `Animal()` — name, breed, kind, and age.

You will however also see some other members — *toString*, *valueOf*, etc — these are defined on *animal* prototype object's prototype object, which is `Object.prototype`.



What happens if you call a method on animal, which is actually defined on Object.prototype?



```
1 animal.valueOf();
```

`valueOf()` returns the value of the object it is called on. In this case, what happens is:

- The browser initially checks to see if the *animal* object has a `valueOf()` method available on it, as defined on its constructor, `Animal()`, and it doesn't.
- The browser then checks to see if the *animal's* prototype object has a `valueOf()` method available on it. It doesn't, then the browser checks *animal's* prototype object's prototype object, and it has. So the method is called.

Exercise – 01

Build a object with a prototype function for generating a full name from the person. The user enters the users name and surname.

Exercise – 02

Make an app where the user enters data about cars as long as he wants. A car is defined by its manufacturer, model and horsepower.

Print the entered cars in the console in the format:

manufacturer – model [horsepower]

E.g.

Volvo – xc40 [122]

Exercise – 03

Make an app where the user enters data about people till he wants to. A person is defined with her name, surname and age. After the input ends, in the console the user can see all people that are older than 18 years.

What is really *this*?

One of the most confusing things in Javascript is this: ***this***

The ***this*** keyword behaves differently in JavaScript compared to most of the other programming languages.

In JavaScript the value of ***this*** not refer to the function in which it is used or it's scope but is determined mostly by the invocation context of function (context.function()) and where it is called.

Let's have a look in four main case examples

Case 1: Default binding



```
1  var a = 5;  
2  
3  function someFunction() {  
4      console.log(this) // Window  
5      console.log(this.a) // 5  
6  }  
7  
8  someFunction();
```

The a variable was defined in the window context so the output of the someFunction() call will be 5 because this will refer to window context which it's the default context.

Case 2: Implicit binding



```
1  var obj = {  
2      a: 4,  
3      someFunction: function () {  
4          console.log(this.a)  
5      }  
6  };  
7  
8  obj.someFunction(); // 4
```

In this case, The object that is standing before the dot is what this keyword will be bound to.

As you see above this will refer to the object obj so the value of this.a will equal 4.

Case 3: Explicit binding

```
1 function helloEarthling() {  
2     console.log(this.name);  
3 }  
4  
5 var arg = 'unused dummy argument'  
6  
7 var person = {  
8     name: 'Marco'  
9 }  
10  
11 helloEarthling.call(person, arg); // Marco
```

In this case, you can force a function call to use a particular object for **this** binding, without putting a property function reference on the object. so we explicitly say to a function what object it should use for this — using functions such as call, apply and bind

```
1 function helloEarthling() {  
2     console.log(this.name);  
3 }  
4  
5 var arg = 'unused dummy argument'  
6  
7 var person = {  
8     name: 'Marco'  
9 }  
10  
11 helloEarthling.apply(person, [arg]); // Marco
```

```
1 function helloEarthling() {  
2     console.log(this.name);  
3 }  
4  
5 var person = {  
6     name: 'Marco'  
7 }  
8  
9 var helloEarthlingCopy = helloEarthling.bind(person); // Marco  
10  
11 helloEarthlingCopy();
```

Case 4: New Binding

```
1  function Person() {
2      /*
3          1- create a new object using the object literal
4          var this = {};
5      */
6
7      // 2- add properties and methods
8      this.name = 'Donald Duck';
9      this.say = function () {
10         return "I am " + this.name;
11     };
12
13     // 3- return this;
14 }
15
16 var name = 'Ahmed';
17 var result = new Person();
18 console.log(result.name); // Donald Duck
```

The function that is called with new operator when the code `new Person(...)` is executed, the following things happen:

- 1- An empty object is created and referenced by **this** variable, inheriting the prototype of the function.
- 2- Properties and methods are added to the object referenced by **this**.
- 3- The newly created object referenced by this is returned at the end

Exercise – 04

Find the bug: you have in front of you a small application for getting the volume of a Cylinder.



```
1  function Cylinder(cylHeight, cylRadius) {  
2      this.cylHeight = cylHeight;  
3      this.cylRadius = cylRadius;  
4  }  
5  
6  Cylinder.prototype.Volume = function () {  
7      return cylHeight * Math.PI * cylRadius * cylRadius;  
8  };  
9  
10 let cyl = new Cylinder(7, 4);  
11 console.log('volume =', cyl.Volume().toFixed(2));
```

Exercise – 05

Create a dice rolling app.

Steps:

1. Create in html one div where you will print the result and one button which will trigger the dice toss
2. Create an Dice objects which has a numberOfSides property
3. Create a method "roll" on the Dice object (prototype)
4. Create a function for printing the number
5. Connect everything and make it work 😊

Enumeration of properties of an object

Enumerable properties are those properties whose internal enumerable flag is set to true, which is the default for properties created via simple assignment or via a property initializer. Properties defined via [Object.defineProperty](#) and such default enumerable to false.

Enumerable properties show up in [for...in](#) loops unless the property's key is a [Symbol](#). Ownership of properties is determined by whether the property belongs to the object directly and not to its prototype chain. Properties of an object can also be retrieved in total.

for...in



```
1  let person = {  
2    name: 'Pete',  
3    surname: 'Sampras',  
4    born: 1971  
5  }  
6  
7  for (let key in person) {  
8    console.log(key, person[key]);  
9  }
```

This method traverses all enumerable properties of an object and its prototype chain.

A for...in loop only iterates over enumerable, non-Symbol properties.

The loop will iterate over all enumerable properties of the object itself and those the object inherits from its prototype chain (properties of nearer prototypes take precedence over those of prototypes further away from the object in its prototype chain).

Object.keys



```
1 let person = {  
2   name: 'Roger',  
3   surname: 'Federer',  
4   born: 1981  
5 }  
6  
7 console.log(Object.keys(person));
```

This method returns an array with all the own (NOT in the prototype chain) enumerable property names (keys) of an object

Parameters

The object of which the enumerable's own properties are to be returned.

Return value

An array of strings that represent all the enumerable properties of the given object.

Object.getOwnPropertyNames()

`Object.getOwnPropertyNames()` returns an array whose elements are strings corresponding to the enumerable and non-enumerable properties found directly in a given object *obj*.

The ordering of the enumerable properties in the array is consistent with the ordering exposed by a for...in loop (or by Object.keys()) over the properties of the object.

According to ES6, the integer keys of the object (both enumerable and non-enumerable) are added in ascending order to the array first, followed by the string keys in the order of insertion.

Object.keys



```
1  let person = {
2    name: 'Donatello'
3  };
4
5  person.age = '57';
6
7  person['country'] = 'Croatia';
8
9  Object.defineProperty(person, 'salary', {
10    value: '100,000 euro',
11    enumerable: false
12  })
13
14  console.log(Object.keys(person));
```

Object.getOwnPropertyNames()



```
1  let person = {
2    name: 'Donatello'
3  };
4
5  person.age = '57';
6
7  person['country'] = 'Croatia';
8
9  Object.defineProperty(person, 'salary', {
10    value: '100,000 euro',
11    enumerable: false
12  })
13
14  console.log(Object.getOwnPropertyNames(person));
```