[Lesson 7]

Roi Yehoshua 2018

[DAN.IT EDUCATION]

What we learnt last time?

- Functions in Javascript
- Passing functions as arguments
- Scope of various types of variables
- Debugging Javascript code

[Our targets for today]

- Basics of Object Oriented Programming
- JavaScript objects
- Object methods
- Object cloning
- ***this*** keyword

[DAN.IT] EDUCATION

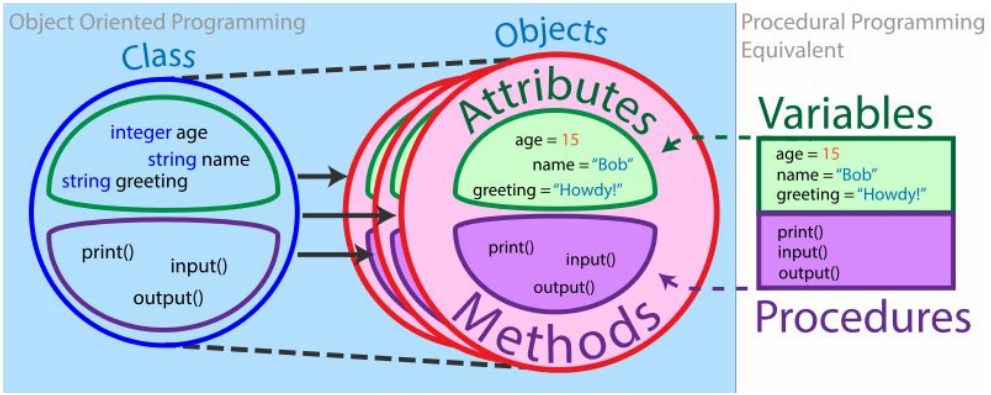
Object Oriented Programming

→ **Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which represent the entities in the program

→ Objects may contain data, in the form of fields, also known as attributes or properties

→ Objects may contain code, in the form of functions, also known as methods

→ Objects are instances of classes, which also determine their type



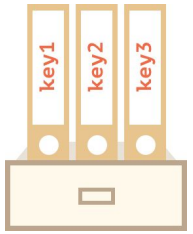
Objects in JavaScript

→ An object is a collection of related data and/or functionality

→ We can imagine an object as a cabinet with signed files

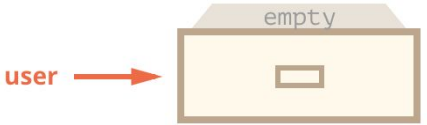
  → Every piece of data is stored in its file by the key

  → It's easy to find a file by its name or add/remove a file



→ An empty object ("empty cabinet") can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user =  {};  // "object literal" syntax
```



→ Usually, the figure brackets {...} are used. That declaration is called an *object literal*.

Literals and Properties

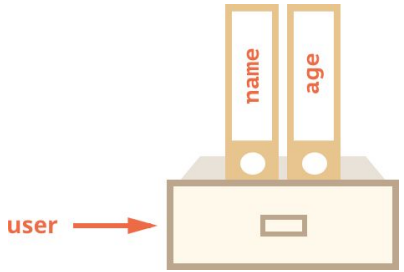
→ We can immediately put some properties into {...} as “key: value” pairs:

```
let user =  {    // an object
    name:  'John',     // by key "name" store value 'John'
    age: 30// by key "age" store value 30
};
```

→ A property has a key (also known as “name” or “identifier”) before the colon ":" and a value to the right of it

→ In the user object, there are two properties:

- → The first property has the name "name" and the value 'John'
- → The second one has the name "age" and the value 30



[DAN.IT] EDUCATION

Exercise (1)

→ Write the following code, one line for each action:

- → Create an empty object product
- → Add the property name with the value ’Laptop’
- → Add the property price with the value 1200
- → Change the value of the price to 1000
- → Show the product’s name and price on the screen
- → Remove the property name from the object

Literals and Properties

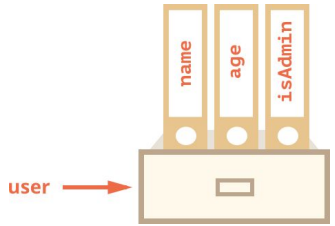
→ Property values are accessible using the **dot notation**:

```
// Get fields of the object
alert(user.name); // John
alert(user.age); // 30
```

→ You can add new properties to an object also using the dot notation

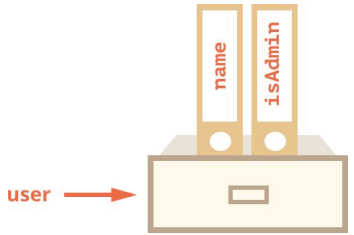
→ The property can be of any type

```
// Add properties to an object
user.isAdmin = true;
```



→ To remove a property, we can use **delete** operator:

```
// Delete properties from an object
delete user.age;
```



[DAN.IT] EDUCATION

Literals and Properties

→ Square brackets also provide a way to obtain the property name as the result of any expression, which may depend on user input as in the following example:

```
let user = {
     name: "John",
     age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// access by variable
alert(user[key]); // John (if enter "name")
```

→ Square brackets are much more powerful than the dot notation, but they are also more cumbersome to write

→ So usually when property names are known and simple, the dot is used. And if we need something more complex, then we switch to square brackets.

Property Value Shorthand

→ The use-case of making a property from a variable with the same name is so common, that there's a special *property value shorthand* to make it shorter:

```
let name = "John";
let age = 30;

let user  = {
     name: name,
     age: age
};
alert(user.name); // John
```

```
let name = "John";
let age = 30;

let user = {
     name, // same as name: name
     age // same as age: age
};
alert(user.name); // John
```

→ We can use both normal properties and shorthands in the same object:

```
let user = {
     name,  // same as name: name
     age: 30
};
```

Property Existence Check

→ Accessing a non-existing property returns undefined

→ Thus, you can test if a property exists by getting it and comparing it to undefined:

```
let user = {};

alert(user.noSuchProperty === undefined); // true means "no such property"
```

→ There also exists a special operator in to check for the existence of a property

```
let user = { name: "John", age: 30 };

alert("age" in user); // true, user.age exists
alert("blabla" in user); // false, user.blabla doesn't exist
```

[DAN.IT] EDUCATION

for..in Loop

→ To iterate over all keys of an object, there exists a special form of the loop: for..in

→ The syntax:

```
for (key in object) {
     // executes the body for each key among object properties
}
```

→ For instance, let's output all properties of user:

```
let user = {
    name: "John",
    age: 30,
    isAdmin: true
};

for (let key in user) {
    // keys
    alert(key);  // name, age, isAdmin
    // values for the keys
    alert(user[key]); // John, 30, true
}
```

[DAN.IT] EDUCATION

Exercise (2)

→ We have an object storing salaries of our team:

```
let salaries = {
    John: 100,
    Ann: 160,
    Peter: 130
}
```

→ Write the code to sum all salaries and store in the variable sum

→ Should be 390 in the example above

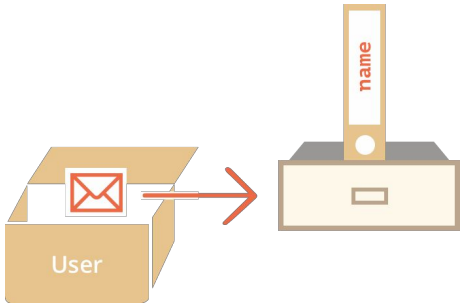
→ If salaries is empty, then the result must be 0

[DAN.IT] EDUCATION

Object References

→ When you define an object in JavaScript, the object's variable stores not the object itself, but its address in memory, i.e., a reference to it

```
let user = {
    name: "John"
};
```



→ The object is stored somewhere in memory, and the variable user has a "reference" to it

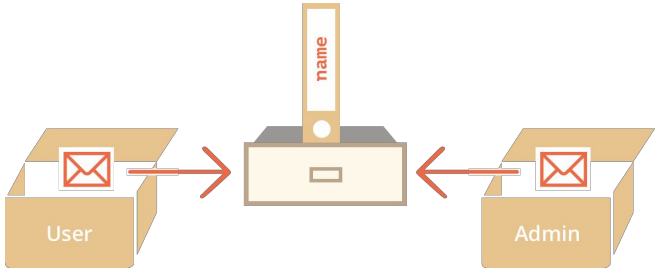
→ If we imagine an object as a cabinet, then a variable is a key to it

Copying Objects by Reference

→ When an object variable is copied (or passed as a function argument) – the reference is copied, not the object:

```
let user = { name: "John" };

let admin = user; // copy the reference
```



→ Now we have two variables, each one with the reference to the same object

→ We can use any variable to access the cabinet and modify its contents:

```
admin.name = "David"; // changed by the "admin" reference
alert(user.name); // "David", changes are seen from the "user" reference
```

[DAN.IT] EDUCATION

Comparing Objects by Reference

→ The equality == and strict equality === operators for objects work exactly the same

→ Two objects are equal only if they are the same object

→ For instance, two variables reference the same object, they are equal:

```
let a = {};
let b = a; // copy the reference

alert(a == b); // true, both variables reference the same object
alert(a === b); // true
```

→ And here two independent objects are not equal, even though both are empty:

```
let a = {};
let b = {}; // two independent objects

alert(a == b); // false
```

[DAN.IT] EDUCATION

Const Object

→ An object declared as const *can* be changed

→ For instance:

```
const user = {
    name: "John"
};
user.age = 25; // No error, since the reference "user" doesn't change
alert(user.age); // 25
```

→ The const would give an error if we try to set user to something else, for instance:

```
// Error (can't reassign user)
user = {
    name: "David"
};
```

Cloning Objects

- → Copying by reference is good most of the time
- → However, there are we cases where we want to duplicate an existing object, i.e., create an independent copy of it
- → We can write the code that replicates the structure of an existing object by iterating over its properties and copying them on the primitive level (using a for..in loop)
- → Also we can use the method Object.assign() for that. Its syntax is:

```
Object.assign(dest[, src1, src2, src3...])
```

  - → Arguments dest, and src1, ..., srcN (can be as many as needed) are objects
  - → It copies the properties of all objects src1, ..., srcN into dest
  - → Then it returns dest

[DAN.IT EDUCATION]

Cloning Objects

→ For example, we can copy all properties of user into an empty object

```
let user = {
     name: "John",
     age: 30
};
let clone = Object.assign({}, user);

// now clone is a fully independent clone
clone.name = "David"; // changed the data in it
alert(user.name); // still John in the original object
```

→ Note that if there are properties in user that are references to other objects, they will be copied by reference

→ There is a standard algorithm for **deep cloning** that handles such cases

→ You can use a working implementation of it from the JavaScript library lodash

→ The method is called _.cloneDeep(obj).

Object Methods

→ JavaScript methods are actions that can be performed on objects

→ A method is a property containing a function definition:

```
let user = {
    name: "John",
    age: 30
};
user.sayHi = function () {
    alert("Hello!");
};

user.sayHi(); // Hello!
```

→ Here we've used a Function Expression to create the function and assign it to the property user.sayHi of the object

→ So, here we've got a method sayHi of the object user

Method Shorthand

→ There exists a shorter syntax for methods in an object literal:

```
let user = {
    sayHi: function () {
        alert("Hello");
    }
};
```

→ we can omit "function" and just write sayHi():

```
let user = {
    sayHi() { // same as "sayHi: function()"
        alert("Hello");
    }
};
```

Exercise (3)

→ Create an object calculator with three methods:

- → read() prompts for two values and saves them as object properties
- → sum() returns the sum of saved values
- → mul() multiplies saved values and returns the result

```
let calculator = {
    // ... your code ...
};

calculator.read();
alert(calculator.sum());
alert(calculator.mul());
```

[DAN.IT] EDUCATION

Symbol Type

→ Symbol is a primitive type for unique identifiers

→ Object property keys may be either of string type, or of symbol type

  → Property keys of other types are coerced to strings

→ A value of Symbol type can be created using Symbol():

```
let id = Symbol();
```

→ We can also give symbol a description, mostly useful for debugging purposes:

```
// id is a symbol with the description "id"
let id = Symbol("id");
```

→ Symbols are guaranteed to be unique, even if they have the same description

```
let id1 = Symbol("id");
let id2 = Symbol("id");
alert(id1 == id2); // false
```

[DAN.IT] EDUCATION

Hidden Properties

→ Symbols allow us to create “hidden” properties of an object, that no other part of code can occasionally access or overwrite

→ For instance, if we want to store an “identifier” for the object user, we can use a symbol as a key for it:

```
let user = {
      name: "John"
};
let id = Symbol("id");

user[id] =  "ID Value";
alert(user[id]); // we can access the data using the symbol as the key
```

→ The benefit of using Symbol("id") over a string "id" is that if another script wants to have its own "id" property inside user for its own purposes, then it can create its own Symbol("id"), without causing any conflicts:

```
// ...
let id = Symbol("id");
user[id] = "Their id value";
```

[DAN.IT] EDUCATION

Symbol Type

→ If we want to use a symbol in an object literal, we need square brackets:

```
let id = Symbol("id");

let user = {
    name: "John",
    [id]: 123 // not just "id: 123"
};
```

→ That's because we need the value from the variable id as the key, not the string "id"

→ Symbols are skipped by for..in loops

→ In contrast, Object.assign copies both string and symbol properties

System Symbols

- There exist many “system” symbols that JavaScript uses internally, and we can use them to fine-tune various aspects of our objects
- They are listed in the specification in the Well-known symbols table, e.g.:
  - Symbol.hasInstance
  - Symbol.iterator
  - Symbol.toPrimitive
  - …and so on
- For instance, Symbol.iterative allows us to make an object iterable
  - We’ll see its use very soon

Constructor Functions

- → The regular {...} syntax allows to create one object
- → But often we need to create many similar objects, like multiple users or menu items
- → That can be done using constructor functions and the "new" operator
- → Constructors are regular functions, that follow two conventions:
  - → They are named with capital letter first
  - → They should be executed only with "new" operator

```
function User(name) {
    this.name = name;
    this.isAdmin = false;
}

let user = new User("Adam");
alert(user.name); // Adam
alert(user.isAdmin); // false

let user2 = new User("Ann");
alert(user.name); // Ann
```

The new operator

→ When a function is executed as new User(...), it does the following steps:

→ A new empty object is created and assigned to **this**

→ The function body executes

→ Usually it modifies this, adds new properties to it

→ The value of this is returned

→ In other words, new User(...) does something like:

```
function User(name) {
    // this =  {};     (implicitly)

    // add properties to this
    this.name = name;
    this.isAdmin = false;

    // return this;    (implicitly)
}
```

Methods in Constructor

→ In the constructor, we can add to **this** not only properties, but methods as well

→ For instance, new User(name) below creates an object with the given name and the method sayHi:

```
function  User(name) {
    this.name  = name;

    this.sayHi = function () {
        alert("My name is: " + this.name);
    };
}

let john = new User("John");
john.sayHi(); // My name is: John
```

Exercise (4)

→ Create a constructor function **Calculator** that creates objects with 3 methods:

- → read() asks for two values using prompt and remembers them in object properties
- → sum() returns the sum of these properties
- → mul() returns the multiplication product of these properties

→ For instance:

```
let calculator = new Calculator();
calculator.read();

alert("Sum = " + calculator.sum());
alert("Mul = " + calculator.mul());
```

[DAN.IT] EDUCATION

Exercise (5)

→ Create a constructor function Accumulator(startingValue)

→ Object that it creates should:

→ Store the “current value” in the property value. The starting value is set to the argument of the constructor startingValue.

→ The read() method should use prompt to read a new number and add it to value.

→ In other words, the value property is the sum of all user-entered values with the initial value startingValue.

→ Here’s the demo of the code:

```
let accumulator = new Accumulator(1); // initial value 1
accumulator.read(); // adds the user-entered value
accumulator.read(); // adds the user-entered value
alert(accumulator.value); // shows the sum of these values
```

[DAN.IT] EDUCATION

Exercise (6)

→ Create a constructor function Calculator that creates “extendable” calculator objects

→ First, implement the method calculate(str) that takes a string like "1 + 2" in the format “NUMBER operator NUMBER” (space-delimited) and returns the result. Should understand plus + and minus -.

```
let calc = new Calculator();
alert(calc.calculate("3 + 7")); // 10
```

→ Then add the method addOperator(name, func) that teaches the calculator a new operation. It takes the operator name and the two-argument function func(a,b) that implements it. Usage example:

```
let powerCalc = new Calculator();
powerCalc.addMethod("*", (a, b) => a * b);
powerCalc.addMethod("/", (a, b) => a / b);
powerCalc.addMethod("**", (a, b) => a ** b);

let result = powerCalc.calculate("2 ** 3");
alert(result); // 8
```

[DAN.IT] EDUCATION

The **this** Keyword

→ It is common that an object method needs to access the information stored in the object to do its job

→ For example, the code inside user.sayHi() may need the name of the user

→ To access the object, a method can use the **this** keyword

→ The value of **this** is the object “before the dot”, i.e., the object that was used to call the method

```
let user  = {
    name: "John",
    age: 30,

    sayHi() {
        alert(this.name); // this == user
    }
};
user.sayHi(); // John
```

Unbounded **this**

→ In JavaScript **this** is “free”, its value is evaluated at call-time and does not depend on where the method was declared, but rather on what’s the object “before the dot”

→ For example, there is no syntax error in a code like this:

```
Function saySomething(){
      alert(this);
}

saySomething(); // undefined (in strict mode)
```

→ In this case **this** is undefined in strict mode

  → If we try to access this.name, there will be an error

→ In non-strict mode (if one forgets use strict) the value of **this** in such case will be the *global object* (window in a browser)

  → This is a historical behavior that "use strict" fixes

[DAN.IT] EDUCATION

this in Arrow Functions

→ Arrow functions are special: they don't have their "own" **`this`**

→ If we reference **this** from such a function, it's taken from the outer "normal" function

→ For instance, here arrow() uses **this** from the outer user.sayHi() method:

```
let user = {
    firstName: "Roi",
    sayHi() {
        let func = () => alert(this.firstName);
        func();
    }
};

user.sayHi(); // Roi
```

Exercise (7)

→ Here the function makeUser() returns an object

→ What is the result of accessing its ref? Why?

```
function makeUser() {
    return {
        name: "John",
        ref: this
    };
};
let user = makeUser();

alert(user.ref.name); // What's the result?
```

Exercise (8)

→ There's a ladder object that allows to go up and down:

```
let ladder = {
    step: 0,
    up() {
        this.step++;
    },
    down() {
        this.step--;
    },
    showStep: function () { // shows the current step
        alert(this.step);
    }
};
```

→ Now, if we need to make several calls in sequence, can do it like this:

```
ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1
```

→ Modify the code of up and down to make the calls chainable, like this:

```
ladder.up().up().down().showStep(); // 1
```

[DAN.IT] EDUCATION

Control questions

1. What is Object Oriented Programming?
2. What is Object?
3. What are the ways to create an object in JavaScript?
4. What is method? How can we call one?
5. Can there be two methods with same name in an object?
6. How can we clone an object?
7. How does ***this*** keyword work?