Cheat Sheet – Objects

JavaScript allows you to write object-orientated code and objects are indeed a key element of the JavaScript language.

Creating Objects

There are different ways to create an object. Probably the easiest one is this one:

```
var myObj = {
    value1: 'a value',
    fn: function() {…}
}
```

But you may also create objects using this syntax:

```
var country = new Object();
country.name = 'Italy';
```

Or this one:

```
var house = Object.create(null);
house.size = 45;
```

There even is another one, you'll see it in the "Constructor Functions" section.

Why do you need that many ways to create objects?
Some ways provide certain advantages, others are there for historic reasons. The first way shown ("literal notation") equals the second one (new Object). But the first one is more expressive and allows you to create an object with fields in one single line (if you don't care about code formatting). Therefore it's preferable to the second way.

Object.create() offers the advantage, that you can pass the prototype of the to-be-created-object as an argument (or null, if no prototype should be set). You'll learn more about Prototypes in this document, but that is a powerful feature which allows you to overwrite JavaScript's default behavior.

Object Literal Notation

The first example for creating an object showed the "Literal Notation" of objects. That's basically a convenient way to create JS objects in a very expressive way.

Indeed, it is that expressive, that JSON (JavaScript Object Notation) became one of the most popular encoding protocols for transferring data (from server to client for example).

Prototypes

Prototypes are objects on which other JavaScript objects are based. It's JavaScript's form of inheritance between objects.
Prototypes are important because they provide functionality to an object, which the object itself might not have.
For example, a prototype may have a greet() method. An object based on that prototype might not have that, but you could still execute obj.greet() since JavaScript will refer to the Prototype (and then the Prototype's Prototype and so on) to find properties or methods accessed by the code.

By default, objects created via literal notation or new Object() inherit from the Object.prototype. You can think of this as JavaScript's default prototype. It ships some useful methods like toString() to convert the object to a string which can be printed to the console for example.

You can set your own Prototype via Object.create(prototypeToBeSet) or constructor functions (see next section).

Learn more about Prototypes in this great article:
http://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/

Constructor Functions

Constructor functions allow you to create your own objects and prototypes. It's best to simply see it in action:

```
var Person = function() {
    this.name = '';
    this.greet = function() {
        console.log('Hi, my name is ' + this.name +
' and I am ' + this.age + ' years old!');
    }
};
var max = new Person();
```

As you see, you simply create a normal function and then you may use this as a constructor for your own objects. Might look strange, but that's how it works in JavaScript.

You set properties for your object by using the *this* keyword in the function (e.g. this.name).

Objects based on that constructor (created, using *new*) have all the properties and methods set up in the function.

The prototype of an object created via a constructor is <function-name>.prototype. For example: Person.prototype.

You can add properties and methods to this prototype as well. This allows you to extend your base object without touching the constructor function.

Learn more about constructor functions here:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor

This

The this keyword is *very* important in JavaScript. As you saw in the constructor section, it allows you to add properties to an object. And in general, it there referred to the object itself.

In line with the above statement, in most cases *this* simply refers to the following: The object executing the code in which *this* is called.

You can control the value of *this* with the bind(), call() and apply() methods. These methods allow you to overwrite the default *this* value.
Read this great answer to understand how they work and differ:
http://stackoverflow.com/a/31922712

Learn more about *this* here:
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/this

defineProperty()

Learn more about available configurations here:
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

Object Methods & Loops

There are some useful built-in methods you may use in conjunction with objects.

For example, you can verify if an object was created with a specific constructor with the *instanceOf* keyword.

```
if (rect1 instanceof Rectangle) {…}
```

You can also check if an object has a certain property with the *in* keyword.

```
if ('width' in rect1) {…}
```

Or you can delete a property using the *delete* keyword.

```
delete rect1.width;
```

Learn more about Object methods here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

If you want to loop through all fields/ properties an object has, you can use the for … of loop:

```
for (field in rect1) {
    console.log(field + ': ' + rect1[field]);
}
```

Learn more about objects here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects
And here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object