Slide 0

Hi everyone. My name is Lina and I'm going to have a small talk about inheritance in JS.

Slide 1

Inheritance is an important concept of object-oriented programming. There is a classical inheritance in such languages as C++, C#, JAVA. It means that classes inherit from other classes and create subclass relationships. But JS can be a bit confusing for developers experienced in object-oriented languages as it is dynamic and does not provide a class implementation per se (even though the class keyword is introduced in ES2015, it is just a syntactical sugar).

When it comes to inheritance, JavaScript has one construction: objects. It means that objects inherit directly from other objects.

Slide 2

First of all, let's take a closer look at such abstracts as "object" and "prototype" due to their importance.

What is an object? Object is an unordered dynamic collection of properties (key-value pairs), where a key is always string and value can be of any JS type: primitive, object, function.

Slide 3

Everything in JS is an object. Every object has a special hidden property [[Prototype]], that is either null or references another object called "prototype".

Slide 4

What is prototype?

Technically it is a regular object created by the JS environment.

There are a lot of methods to manipulate [[Prototype]].

Firstly, let's take a look at the oldest way of setting [[Prototype]] for objects created via constructor function.

Slide 5

Constructor functions are regular functions with two conventions:

-they are named with a capital letter;

-they should be executed only with "new" operator.

On the slide you see a simple constructor function implementation and what happens when it's executed.

Slide 6

As I mentioned before every function has a Function.prorotype property. Please pay attention that it means regular PROPERTY of function named prototype, it is NOT “prototype” (“object prototype”).

Slide 7

By default, all functions have F. prototype set to a regular object created by the JS environment with the only “constructor” property in it. When the constructor function is called “new” operator assigns this object as [[Prototype]] of the newly created instances. You may see a confirmation of my words on this slide. I created a new object john via constructor function User. You can mention that john’s prototype and the User.prototype property are the same objects.

Slide 8

On this slide I created a new function “func” and set it to the User.prototype property. Then I created a new object bill. You see that bill has a new “func” as prototype while previously created john still has a default object as prototype. It means that as soon as object was created there is no connection between F. prototype and the object.

Slide 9

If we change nothing, the constructor property will be available to all users through [[Prototype]]. On this slide I created a new object jack using the john’s constructor.

Slide 10

When I said “change nothing” I meant the following case: constructor exists in the default prototype function’s property, but if we replace the entire default object, there will be no constructor in it (like on the first picture). In order to keep the right constructor, we have to add/remove properties to the default object instead of overwriting it (like on the second one).

Slide 11

The prototype property is widely used by the JS core. All built-in constructor functions (Object, Function, Array, Number, etc.) use it. They set [[Prototype]] of the newly created instances to the corresponding native prototypes according to the rules explained before.

Slide 12

The native prototypes are huge objects with lots of methods. All object’s [[Prototype]] links (the red arrows on the picture) are called “prototype chain”.

Slide 13

Now let’s come back to the inheritance. When we trying to access a property of an object, the property will not only be searched in the object but also in the prototype of the object, in the prototype of the prototype and so on until either a property with a matching name is found or the end of the prototype chain is reached. Such thing is called “prototypal inheritance”.

Slide 14

On this slide you see the inheritance implementation taken from the MDN. There is a function constructor “f” and its instance object “o”. “O” has properties “a” equals to 1 and “b” equals to 2. Then we add properties “b” and “c” (equals to 3 and 4 respectively) to the f. prototype. The full prototype chain looks like object with a, b; object with b, c; Object. prototype and null.

Slide 15

Then we want to log different properties. Is there an “a” property in object “o”? Yes, and its value is 1. Is there a “b” property in object “o”? Yes, and its value is 2. Even though the object prototype also has a “b” property, the JS interpreter wouldn’t visit it. This is called “property shadowing”. Is there a “c” property in object “o”? No, and interpreter will visit o.[[Prototype]] and log value 4. Is there a “d” property in object “o”? No. There is also no property “d” in the whole prototype chain and as the result we have undefined.

Slide 16

The methods are inherited by the same mechanism. The only point I want to pay your attention to there is if we call a method, and the method is taken from the prototype (like the method “m” in the example is taken from the object “o”), **this** still references the current object.

Slide 17

At the end I want to talk about the ECMAScript standard of accessing the object prototype. Following it, the notation someObject.[[Prototype]] is used to designate the prototype of someObject. Since ECMAScript 2015, in order to deal with the prototype, the following methods should be used. Pay attention that the JS property __proto__now is non-standard and shouldn’t be used.