28th December 2019

Variable scope

JavaScript is a very function-oriented language. It gives us a lot of freedom. A function can be created dynamically, passed as an argument to another function and called from a totally different place of code later.

We already know that a function can access variables outside of it.

Now let's expand our knowledge to include more complex scenarios.

We'll talk about `let/const` variables here

In JavaScript, there are 3 ways to declare a variable: `let`, `const` (the modern ones), and `var` (the remnant of the past).

- In this article we'll use `let` variables in examples.
- Variables, declared with `const`, behave the same, so this article is about `const` too.
- The old `var` has some notable differences, they will be covered in the article The old "var".

Code blocks

If a variable is declared inside a code block `{...}`, it's only visible inside that block.

For example:

```
{
  // do some job with local variables that should not be seen outside

  let message = "Hello"; // only visible in this block

  alert(message); // Hello
}

alert(message); // Error: message is not defined
```

We can use this to isolate a piece of code that does its own task, with variables that only belong to it:

```
{
  // show message
  let message = "Hello";
  alert(message);
}

```

```
7  {
8    // show another message
9    let message = "Goodbye";
10   alert(message);
11 }
```

There'd be an error without blocks

Please note, without separate blocks there would be an error, if we use `let` with the existing variable name:

```
1 // show message
2 let message = "Hello";
3 alert(message);
4
5 // show another message
6 let message = "Goodbye"; // Error: variable already declared
7 alert(message);
```

For `if`, `for`, `while` and so on, variables declared in `{...}` are also only visible inside:

```
1 if (true) {
2   let phrase = "Hello!";
3
4   alert(phrase); // Hello!
5 }
6
7 alert(phrase); // Error, no such variable!
```

Here, after `if` finishes, the `alert` below won't see the `phrase`, hence the error.

That's great, as it allows us to create block-local variables, specific to an `if` branch.

The similar thing holds true for `for` and `while` loops:

```
1 for (let i = 0; i < 3; i++) {
2   // the variable i is only visible inside this for
3   alert(i); // 0, then 1, then 2
4 }
5
6 alert(i); // Error, no such variable
```

Visually, `let i` is outside of `{...}`. But the `for` construct is special here: the variable, declared inside it, is considered a part of the block.

Nested functions

A function is called "nested" when it is created inside another function.

It is easily possible to do this with JavaScript.

We can use it to organize our code, like this:

```
function sayHiBye(firstName, lastName) {

  // helper nested function to use below
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

Here the *nested* function `getFullName()` is made for convenience. It can access the outer variables and so can return the full name. Nested functions are quite common in JavaScript.

What's much more interesting, a nested function can be returned: either as a property of a new object or as a result by itself. It can then be used somewhere else. No matter where, it still has access to the same outer variables.

Below, `makeCounter` creates the "counter" function that returns the next number on each invocation:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Despite being simple, slightly modified variants of that code have practical uses, for instance, as a random number generator to generate random values for automated tests.

How does this work? If we create multiple counters, will they be independent? What's going on with the variables here?

Undestanding such things is great for the overall knowledge of JavaScript and beneficial for more complex scenarios. So let's go a bit in-depth.

Lexical Environment

Here be dragons!

The in-depth technical explanation lies ahead.

As far as I'd like to avoid low-level language details, any understanding without them would be lacking and incomplete, so get ready.

For clarity, the explanation is split into multiple steps.

Step 1. Variables

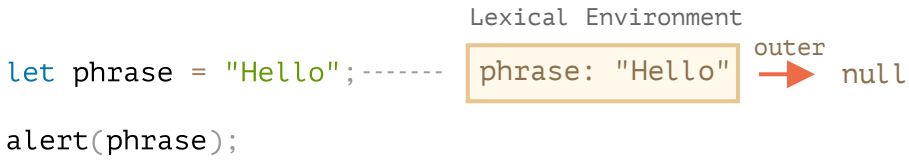
In JavaScript, every running function, code block `{...}`, and the script as a whole have an internal (hidden) associated object known as the *Lexical Environment*.

The Lexical Environment object consists of two parts:

1. *Environment Record* – an object that stores all local variables as its properties (and some other information like the value of `this`).
2. A reference to the *outer lexical environment*, the one associated with the outer code.

A "variable" is just a property of the special internal object, `Environment Record`. "To get or change a variable" means "to get or change a property of that object".

In this simple code without functions, there is only one Lexical Environment:

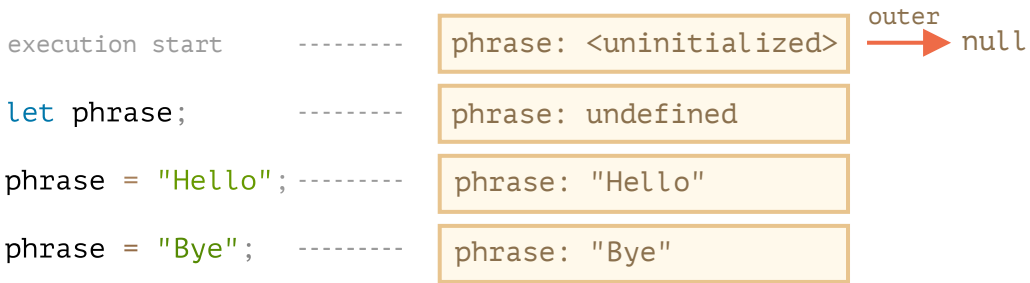


This is the so-called *global* Lexical Environment, associated with the whole script.

On the picture above, the rectangle means Environment Record (variable store) and the arrow means the outer reference. The global Lexical Environment has no outer reference, that's why the arrow points to `null`.

As the code starts executing and goes on, the Lexical Environment changes.

Here's a little bit longer code:



Rectangles on the right-hand side demonstrate how the global Lexical Environment changes during the execution:

1. When the script starts, the Lexical Environment is pre-populated with all declared variables.

- Initially, they are in the “Uninitialized” state. That’s a special internal state, it means that the engine knows about the variable, but won’t allow to use it before `let`. It’s almost the same as if the variable didn’t exist.

2. Then `let phrase` definition appears. There’s no assignment yet, so its value is `undefined`. We can use the variable since this moment.
3. `phrase` is assigned a value.
4. `phrase` changes the value.

Everything looks simple for now, right?

- A variable is a property of a special internal object, associated with the currently executing block/function/script.
- Working with variables is actually working with the properties of that object.

> **Lexical Environment is a specification object**
>
> “Lexical Environment” is a specification object: it only exists “theoretically” in the language specification to describe how things work. We can’t get this object in our code and manipulate it directly.
>
> JavaScript engines also may optimize it, discard variables that are unused to save memory and perform other internal tricks, as long as the visible behavior remains as described.

Step 2. Function Declarations

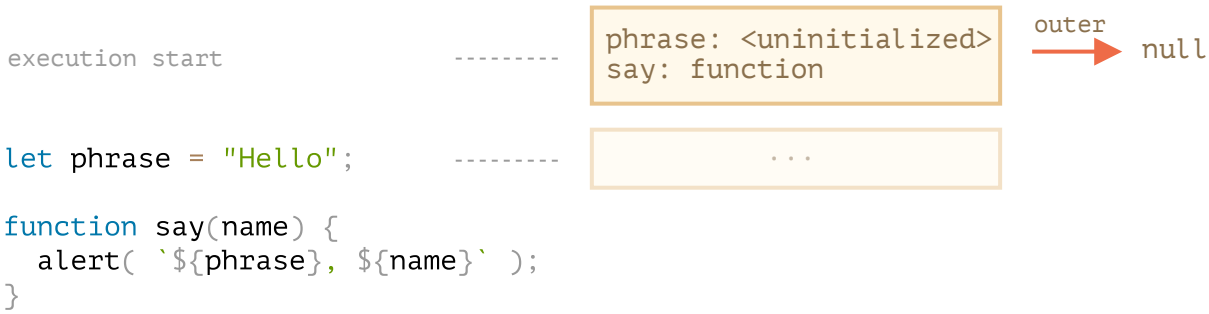
A function is also a value, like a variable.

The difference is that a Function Declaration is instantly fully initialized.

When a Lexical Environment is created, a Function Declaration immediately becomes a ready-to-use function (unlike `let`, that is unusable till the declaration).

That’s why we can use a function, declared as Function Declaration, even before the declaration itself.

For example, here’s the initial state of the global Lexical Environment when we add a function:

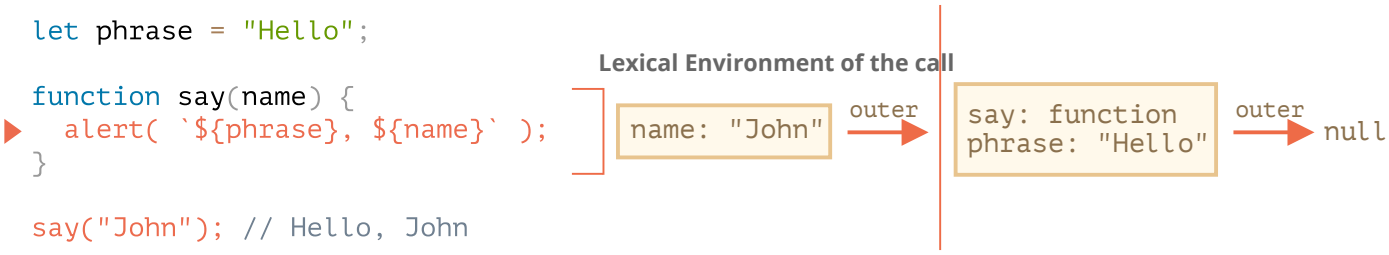


Naturally, this behavior only applies to Function Declarations, not Function Expressions where we assign a function to a variable, such as `let say = function(name)...`.

Step 3. Inner and outer Lexical Environment

When a function runs, at the beginning of the call, a new Lexical Environment is created automatically to store local variables and parameters of the call.

For instance, for `say("John")`, it looks like this (the execution is at the line, labelled with an arrow):



During the function call we have two Lexical Environments: the inner one (for the function call) and the outer one (global):

- The inner Lexical Environment corresponds to the current execution of `say`. It has a single property: `name`, the function argument. We called `say("John")`, so the value of the `name` is `"John"`.
- The outer Lexical Environment is the global Lexical Environment. It has the `phrase` variable and the function itself.

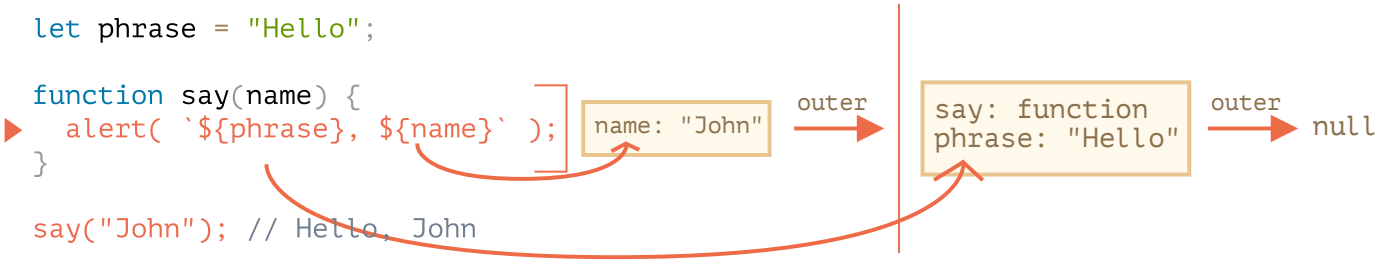
The inner Lexical Environment has a reference to the `outer` one.

When the code wants to access a variable – the inner Lexical Environment is searched first, then the outer one, then the more outer one and so on until the global one.

If a variable is not found anywhere, that's an error in strict mode (without `use strict`, an assignment to a non-existing variable creates a new global variable, for compatibility with old code).

In this example the search proceeds as follows:

- For the `name` variable, the `alert` inside `say` finds it immediately in the inner Lexical Environment.
- When it wants to access `phrase`, then there is no `phrase` locally, so it follows the reference to the outer Lexical Environment and finds it there.



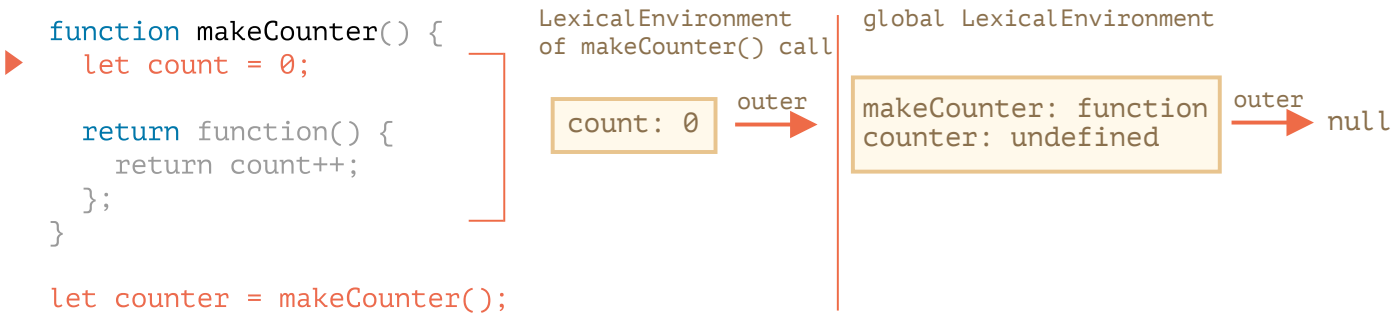
Step 4. Returning a function

Let's return to the `makeCounter` example.

```
1  function makeCounter() {
2    let count = 0;
3
4    return function() {
5      return count++;
6    };
7  }
8
9  let counter = makeCounter();
```

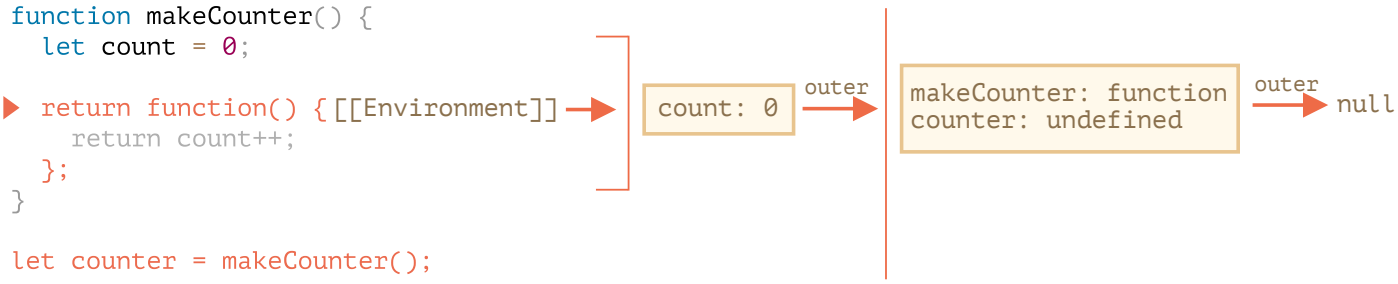
At the beginning of each `makeCounter()` call, a new Lexical Environment object is created, to store variables for this `makeCounter` run.

So we have two nested Lexical Environments, just like in the example above:



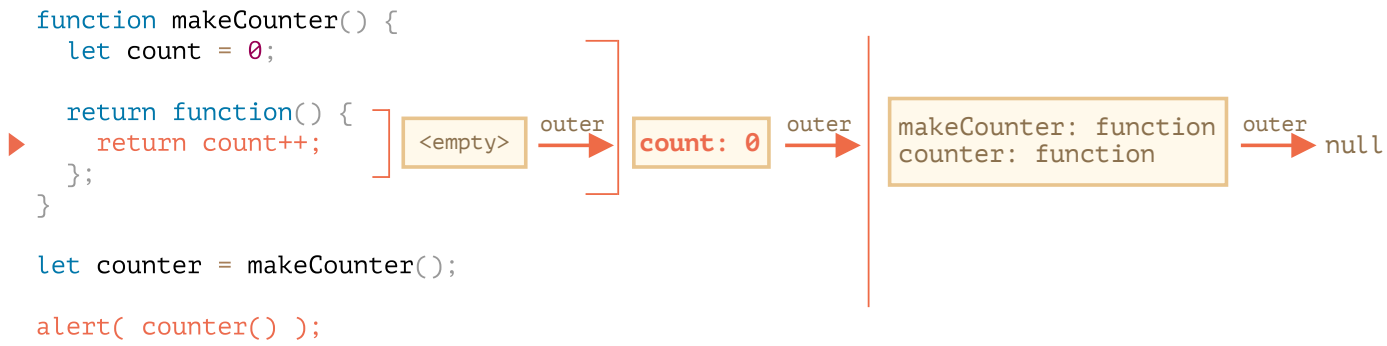
What's different is that, during the execution of `makeCounter()`, a tiny nested function is created of only one line: `return count++`. We don't run it yet, only create.

All functions remember the Lexical Environment in which they were made. Technically, there's no magic here: all functions have the hidden property named `[[Environment]]`, that keeps the reference to the Lexical Environment where the function was created:



So, `counter.[[Environment]]` has the reference to `{count: 0}` Lexical Environment. That's how the function remembers where it was created, no matter where it's called. The `[[Environment]]` reference is set once and forever at function creation time.

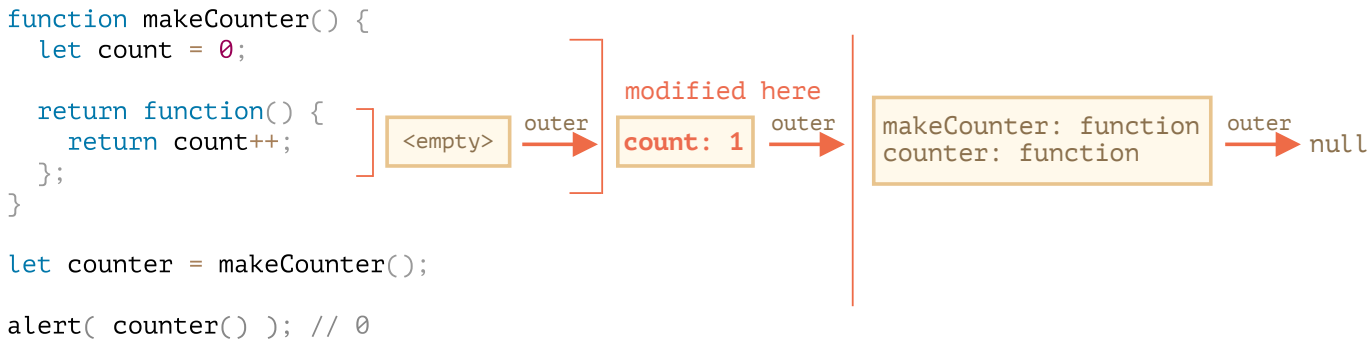
Later, when `counter()` is called, a new Lexical Environment is created for the call, and its outer Lexical Environment reference is taken from `counter.[[Environment]]`:



Now when the code inside `counter()` looks for `count` variable, it first searches its own Lexical Environment (empty, as there are no local variables there), then the Lexical Environment of the outer `makeCounter()` call, where finds it and changes.

A variable is updated in the Lexical Environment where it lives.

Here's the state after the execution:



If we call `counter()` multiple times, the `count` variable will be increased to `2`, `3` and so on, at the same place.

Closure

There is a general programming term "closure", that developers generally should know.

A closure is a function that remembers its outer variables and can access them. In some languages, that's not possible, or a function should be written in a special way to make it happen. But as explained above, in JavaScript, all functions are naturally closures (there is only one exception, to be covered in The "new Function" syntax).

That is: they automatically remember where they were created using a hidden `[[Environment]]` property, and then their code can access outer variables.

When on an interview, a frontend developer gets a question about "what's a closure?", a valid answer would be a definition of the closure and an explanation that all functions in JavaScript are closures, and maybe a few more words about technical details: the `[[Environment]]` property and how Lexical Environments work.

Garbage collection

Usually, a Lexical Environment is removed from memory with all the variables after the function call finishes. That's because there are no references to it. As any JavaScript object, it's only kept in memory while it's reachable.

…But if there's a nested function that is still reachable after the end of a function, then it has `[[Environment]]` property that references the lexical environment.

In that case the Lexical Environment is still reachable even after the completion of the function, so it stays alive.

For example:

```
1  function f() {
2    let value = 123;
3
4    return function() {
5      alert(value);
6    }
7  }
8
```

```
let g = f(); // g.[[Environment]] stores a reference to the Lexical Environme
// of the corresponding f() call
```

Please note that if `f()` is called many times, and resulting functions are saved, then all corresponding Lexical Environment objects will also be retained in memory. All 3 of them in the code below:

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}

// 3 functions in array, every one of them links to Lexical Environment
// from the corresponding f() run
let arr = [f(), f(), f()];
```

A Lexical Environment object dies when it becomes unreachable (just like any other object). In other words, it exists only while there's at least one nested function referencing it.

In the code below, after the nested function is removed, its enclosing Lexical Environment (and hence the `value`) is cleaned from memory;

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // while g function exists, the value stays in memory

g = null; // ...and now the memory is cleaned up
```

Real-life optimizations

As we've seen, in theory while a function is alive, all outer variables are also retained.

But in practice, JavaScript engines try to optimize that. They analyze variable usage and if it's obvious from the code that an outer variable is not used – it is removed.

An important side effect in V8 (Chrome, Opera) is that such variable will become unavailable in debugging.

Try running the example below in Chrome with the Developer Tools open.

When it pauses, in the console type `alert(value)`.

```
function f() {
  let value = Math.random();

  function g() {
```

```
    debugger; // in console: type alert(value); No such variable!
  }

  return g;
}

let g = f();
g();
```

As you could see – there is no such variable! In theory, it should be accessible, but the engine optimized it out.

That may lead to funny (if not such time-consuming) debugging issues. One of them – we can see a same-named outer variable instead of the expected one:

```
let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // in console: type alert(value); Surprise!
  }

  return g;
}

let g = f();
g();
```

This feature of V8 is good to know. If you are debugging with Chrome/Opera, sooner or later you will meet it.

That is not a bug in the debugger, but rather a special feature of V8. Perhaps it will be changed sometime. You always can check for it by running the examples on this page.

Tasks

Does a function pickup latest changes?

importance: 5

The function sayHi uses an external variable name. When the function runs, which value is it going to use?

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi(); // what will it show: "John" or "Pete"?
```

Such situations are common both in browser and server-side development. A function may be scheduled to execute later than it is created, for instance after a user action or a network request.

So, the question is: does it pick up the latest changes?

solution

Which variables are available?

importance: 5

The function `makeWorker` below makes another function and returns it. That new function can be called from somewhere else.

Will it have access to the outer variables from its creation place, or the invocation place, or both?

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "John";

// create a function
let work = makeWorker();

// call it
work(); // what will it show?
```

Which value it will show? “Pete” or “John”?

solution

Are counters independent?

importance: 5

Here we make two counters: `counter` and `counter2` using the same `makeCounter` function.

Are they independent? What is the second counter going to show? `0,1` or `2,3` or something else?

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}
```

```
8
9  let counter = makeCounter();
10 let counter2 = makeCounter();
11
12 alert( counter() ); // 0
13 alert( counter() ); // 1
14
15 alert( counter2() ); // ?
16 alert( counter2() ); // ?
```

solution

Counter object

importance: 5

Here a counter object is made with the help of the constructor function.

Will it work? What will it show?

```
1  function Counter() {
2    let count = 0;
3
4    this.up = function() {
5      return ++count;
6    };
7    this.down = function() {
8      return --count;
9    };
10 }
11
12 let counter = new Counter();
13
14 alert( counter.up() ); // ?
15 alert( counter.up() ); // ?
16 alert( counter.down() ); // ?
```

solution

Function in if

Look at the code. What will be the result of the call at the last line?

```
1  let phrase = "Hello";
2
3  if (true) {
4    let user = "John";
5
6    function sayHi() {
7      alert(`${phrase}, ${user}`);
```

```
  }
}

sayHi();
```

solution

Sum with closures

importance: 4

Write function `sum` that works like this: `sum(a)(b) = a+b`.

Yes, exactly this way, using double parentheses (not a mistype).

For instance:

```
sum(1)(2) = 3
sum(5)(-1) = 4
```

solution

Is variable visible?

importance: 4

What will be the result of this code?

```
let x = 1;

function func() {
  console.log(x); // ?

  let x = 2;
}

func();
```

P.S. There's a pitfall in this task. The solution is not obvious.

solution

Filter through function

importance: 5

We have a built-in method `arr.filter(f)` for arrays. It filters all elements through the function `f`. If it returns `true`, then that element is returned in the resulting array.

Make a set of "ready to use" filters:

- `inBetween(a, b)` – between `a` and `b` or equal to them (inclusively).
- `inArray([...])` – in the given array.

The usage must be like this:

- `arr.filter(inBetween(3,6))` – selects only values between 3 and 6.
- `arr.filter(inArray([1,2,3]))` – selects only elements matching with one of the members of `[1,2,3]`.

For instance:

```
1  /* .. your code for inBetween and inArray */
2  let arr = [1, 2, 3, 4, 5, 6, 7];
3
4  alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
5
6  alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Open a sandbox with tests.

solution

Sort by field

importance: 5

We've got an array of objects to sort:

```
1  let users = [
2    { name: "John", age: 20, surname: "Johnson" },
3    { name: "Pete", age: 18, surname: "Peterson" },
4    { name: "Ann", age: 19, surname: "Hathaway" }
5  ];
```

The usual way to do that would be:

```
1  // by name (Ann, John, Pete)
2  users.sort((a, b) => a.name > b.name ? 1 : -1);
3
4  // by age (Pete, Ann, John)
5  users.sort((a, b) => a.age > b.age ? 1 : -1);
```

Can we make it even less verbose, like this?

```
users.sort(byField('name'));
users.sort(byField('age'));
```

So, instead of writing a function, just put `byField(fieldName)`.

Write the function `byField` that can be used for that.

solution

Army of functions

importance: 5

The following code creates an array of `shooters`.

Every function is meant to output its number. But something is wrong…

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
    let shooter = function() { // shooter function
      alert( i ); // should show its number
    };
    shooters.push(shooter);
    i++;
  }

  return shooters;
}

let army = makeArmy();

army[0](); // the shooter number 0 shows 10
army[5](); // and number 5 also outputs 10...
// ... all shooters show 10 instead of their 0, 1, 2, 3...
```

Why do all of the shooters show the same value? Fix the code so that they work as intended.

Open a sandbox with tests.

solution

Previous lesson Next lesson

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)