# Function hoisted

Обратите внимание, что присваивание значений переменным не поднимается вместе с их объявлением. Поднимаются только объявления переменных. В случае с функциями, поднимается вся функция целиком. Существуют два основных способа объявить функцию, давайте их рассмотрим:

function test() {

foo(); *// TypeError "foo is not a function"*

bar(); *// "this will run!"*

var foo = function () { *// функциональное выражение, присвоенное локальной переменной 'foo'*

alert("this won't run!");

}

function bar() { *// объявление функции с именем 'bar'*

alert("this will run!");

}

}

test();

в этом случае поднимается только функция bar. Идентификатор «foo» также поднимается, но не анонимная функция — она остаётся на месте.  
  
Вот мы и описали основные моменты «поднятия» переменных и функций. Конечно, JavaScript не был бы сам собой, если бы не было особых случаев, в которых всё немного сложнее.

# References obj

var obj = { a: 1 };

(function(obj) {

obj = { a: 2 };

})(obj);

console.log(obj.a);

При вызове анонимной функции создастся локальная переменная obj в её области видимости. А затем создаётся новый объект {a : 2}, ссылка на который попадает в локальную переменную obj, но переменная из верхнего скоупа будет всё так же ссылаться на старый объект.

# Контекст вызова

Logger = function(logFn) {

\_logFn = logFn;

this.log = function(message) {

\_logFn(new Date() + ": " + message);

}

}

var logger = new Logger(console.log);

logger.log("Hi!");

logger.log("Wazzup?");

В консоли мы увидим TypeError: Illegal invocation  
А всё потому что при вызове logger.log(), контекст выполнения функции — logger  
  
Чтобы починить можно вспомнить про встроенные методы функций .apply(), .call(), .bind()

< rightLogger = new Logger(console.log.bind(console))

> Logger {log: function}

< rightLogger.log("It's works")

> Sat Oct 04 2014 00:32:49 GMT+0400 (MSK): It's works

(function() {

var x = 1;

function x() {};

console.log(x); })()

Функции объявленный при помощи function declaration имеют больший приоритет и понимаются выше var. Поэтому интерпретатор сначала выполнит function x() {};, а затем var x = 1;  
  
< (function() {  
var x = 1;  
  
function x() {};  
  
console.log(x);   
})()  
  
> undefined  
1

(function(){

return typeof arguments;

})();

The typeof operator with an array or an object returns the string "object".

var f = function g(){ return 23; };

typeof g();

Correct answer: error.

The two methods are almost equivalent. The only difference is that function declaration is hoisting the function definition (moving it to the top of the current scope).

In the quiz question, the function is defined with an assignment statement and not declared.

**g** is the function name, and it's only available in the scope of the function itself.

Since the second line of code in the snippet (typeof g();) is outside of the function body, both g() and typeof g() return an error

(function(x){

delete x;

return x;

})(1);

The logged value in this case is false, and delete only returns false if the property cannot be deleted.

When a function is executed, its arguments become properties of the function's activation object: activation\_object.x = 1. These properties have DontDelete attribute, hence they cannot be deleted

/\* remember that `this` refers to global object when in global scope \*/

var GLOBAL\_OBJECT = this;

var foo = 1;

GLOBAL\_OBJECT.foo; // 1

foo === GLOBAL\_OBJECT.foo; // true

function bar(){}

typeof GLOBAL\_OBJECT.bar; // "function"

GLOBAL\_OBJECT.bar === bar; // true

The only difference is that when in Function code, a Variable object is not a Global object, but a so-called **Activation object**. Activation object is created every time execution context for Function code is entered.

var y = 1, x = y = typeof x;

x;

Correct answer: "undefined".

The = operator in JavaScript has right associativity, which means that x = y = typeof x is equivalent to x = (y = typeof x). First, the expression typeof x is evaluated. Since x has not been defined yet, typeof x evaluates to the string "undefined". After that, y is assigned the value of "undefined", and then x is assigned the same value. As a result, when the second line of code in this snippet is run, x has the value of "undefined" (a string).

(function f(f){

return typeof f();

})(function(){ return 1; });

Correct answer: "number".

The function f(f) {return typeof f();} is immediately invoked with the argument function(){ return 1; }.

var foo = {

bar: function() { return this.baz; },

baz: 1

};

(function(){

return typeof arguments[0]();

})(foo.bar);

Correct answer: "undefined".

var foo = {

bar: function(){ return this.baz; },

baz: 1

}

typeof (f = foo.bar)();

Correct answer: "undefined".

In the original snippet of code, we start by assigning the value of foo.bar to the variable f. As a result, the variable f points to the function function (){ return this.baz; }. However, when this function is invoked, the keyword this no longer refers to foo.

var f = (function f(){ return "1"; }, function g(){ return 2; })();

typeof f;

Correct answer: "number".

In order to understand this snippet of code, we have to look into the comma operator. A good explanation of the subject can be found [here](https://javascriptweblog.wordpress.com/2011/04/04/the-javascript-comma-operator/). According to [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator), "the comma operator evaluates each of its operands (from left to right) and returns the value of the last operand". Here is a simple example:

var y = ("abc", 5);

y; //5

var x = 1;

if (function f(){}) {

x += typeof f;

}

x;

Correct answer: "1undefined".

var x = [typeof x, typeof y][1];

typeof typeof x;

Correct answer: "string".

The original snippet of code can be rewritten in the following way:

var x = ["undefined", "undefined"][1];

typeof typeof x;

(function(foo){

return typeof foo.bar;

})({ foo: { bar: 1 } });

Correct answer: "undefined".

(function f(){

function f(){ return 1; }

return f();

function f(){ return 2; }

})();

Correct answer: 2.

First, when Function A has just been initialized, the identifier f points to Function A. Then the declaration of Function B makes the identifier f, which used to refer to Function A, point to Function B. Finally, the declaration of Function C redefines f and makes it point to Function C.

function f(){ return f; }

new f() instanceof f;

Correct answer: false.

with (function(x, undefined){}) length;

Correct answer: 2.