



ФУНКЦИИ

Содержание

1. Функции	3-7
◦ Вызов функции	4
◦ Параметры функции	5
◦ Возврат значений	6
2. Типы объявления функций	8-9
3. Область видимости	10-11
4. Замыкание	12-16
◦ Поиск переменных	14
5. Анонимные функции	17-18
6. Возврат функций	19-21
7. IIFE	22-23
8. Всплытие	24-30
◦ Всплытие и function expression	29

1

ФУНКЦИИ

Функции — это кусочки JS-кода, которые можно вызывать повторно.

У функции может быть имя и параметры.

Имя функции — имя, по которому можно вызвать функцию.

Параметры — данные, которые могут быть переданы в функцию при ее вызове и использоваться внутри функции.

Аргументы — непосредственно данные, которые передаются в функцию при ее вызове.

Пример объявления функции:

```
function someName() {  
    console.log('hello');  
}
```

Вызов функции

```
someName();
```

Другими словами: вызов функции осуществляется при помощи указания имени функции и круглых скобок.

Обратите внимание на два определения:

- **объявление функции**
- **вызов функции**

Если просто объявить функцию, но не вызвать ее, то код внутри нее никогда не отработает.

Параметры функции

Как указано выше - функция может иметь параметры.

```
function sum(a, b) {  
  console.log(a + b);  
}
```

В данном случае была объявлена функция с именем *sum*. Так же было указано, что функция принимает два параметра - **a** и **b**.

Задача функции - сложить два переданных в нее числа и вывести результат на экран.

Как было так же сказано выше, чтобы воспользоваться функцией, ее надо вызвать. При вызове функции можно передать ей аргументы в качестве параметров:

```
sum(5, 10);
```

Была вызвана функция *sum*, а в качестве параметров **a** и **b** были переданы числа: **5** и **10** соответственно.

Каждый аргумент будет записан в соответствующий параметр функции. Таким образом, переданное число **5** будет доступно через параметр **a**, а число **10** - через параметр **b**.

Повторим еще раз, что **функции** - это кусочки кода, которые могут быть вызваны повторно. Таким образом мы можем вызывать функцию `sum` неограниченное количество раз с разными аргументами и она будет каждый раз выводить разное значение:

```
sum(1, 10);
```

 В консоле отобразится 11

```
sum(10, 20);
```

 В консоле отобразится 30

```
sum(8, 12);
```

 В консоле отобразится 20

```
sum(10, 15);
```

 В консоле отобразится 25

Возврат значений

Функция может возвращать результат своего выполнения.

Что именно возвращать - решает разработчик.

Возврат значения производится при помощи оператора **return**.

```
function sum(a, b) {  
    return a + b;  
}
```

Переписанный вариант функции не будет ничего выводить на экран, а лишь вернет значение.

Получить возвращаемое функцией значение можно просто присвоив вызов функции в переменную:

```
var s = sum(10, 10);
```

 в переменной `s` будет значение 20

Стоит так же отметить, что оператор **return** не только выбрасывает результат работы функции во “внешний мир”, но еще и выходит из самой функции. То есть код, написанный внутри функции, после оператора **return**, никогда не выполнится:

```
function sum(a, b) {  
    return a + b;  
  
    console.log('!!!!');  
}
```

console.log никогда не выполнится, т.к. до того, как интерпретатор доберется до него, выполнится **return** и произойдет выход из функции.

2

Типы объявления функций

Существует два способа объявления функций:

Function Expression и **Function Declaration**

Более глобальную разницу между ними мы рассмотрим позже, а пока посмотрим на два примера:

Function Declaration:

```
function sum(a, b) {  
    return a + b;  
}
```

Function Expression:

```
var sum = function(a, b) {  
    return a + b;  
};
```

Исходя из приведенных примеров, в случае с **Function Declaration**, интерпретатор объявляет переменную с именем функции и присваивает ей описание функции.

В случае с **Function Expression** - мы сами объявляем переменную с именем функции и присваиваем ей описание функции. В таком случае мы можем даже не писать имя функции. Другими словами: если объявление функции является частью какого-либо другого выражения, то такое объявление называется **Function Expression**.

В данном случае, объявление функции является частью выражения по объявлению переменной *sum*.

Не смотря на различия в объявлении функций: ни результат их выполнения, ни способ их вызова не меняется.

Как уже было сказано ранее - мы посмотрим на их отличия позже.

3

Область ВИДИМОСТИ

Область видимости или **scope** - это отрезок кода, в пределах которого мы имеем доступ к какой-либо переменной.

Каждая функция имеет свою область видимости:

```
function func(a) {  
  var b = 10;  
  
  return a + b;  
}
```

Здесь объявляется функция с именем *func*, которая принимает 1 параметр - **a**. Внутри функции объявляется переменная **b**.

То, что создано внутри функции - является часть области видимости этой функции и не может быть использовано вне этой функции.

Так, переменная **b** объявлена внутри функции *func* и является частью ее области видимости, соответственно использовать переменную **b** можно только внутри функции *func*:

```
function func(a) {  
  var b = 10;  
  
  return a + b;  
}  
  
console.log(b); //ошибка! b доступна только внутри func
```

Другими словами: **область видимости** - это набор переменных, которые доступны только внутри этой области.

Кстати, *func* является часть глобальной области видимости.

4

Замыкание

Замыкание - это способность функции запоминать область видимости, в которой эта функция была объявлена.

Посмотрим пример:

```
var b = 10;

function func(a) {
    var c = 100;

    return a + b + c;
}

func(1);
```

Переменная **a** является параметром функции *func*, а параметры функции всегда входят в область видимости функции, соответственно, переменная **a** свободно используется внутри *func* и недоступна извне.

Переменная **b** объявлена в глобальной области видимости. Функция *func* тоже объявлена в глобальной области видимости.

Если замыкание - это способность функции запоминать область видимости, в которой эта функция была объявлена, то функция *func* запомнила, что в момент объявления, в области видимости, в которой она была объявлена, была доступна переменная **b**, соответственно, функция *func* имеет доступ к этой переменной, а значит может использовать ее внутри себя.

Переменная **c** является частью области видимости функции, т.к. была объявлена внутри функции, а значит, может свободно использоваться внутри *func* и недоступна извне.

В приведенном выше примере, функция вернет **111**

Вместо **a** - **1**, вместо **b** - **10**, вместо **c** - **100**: $1 + 10 + 100 = 111$

Поиск переменных

Когда функция пытается использовать переменную, то сначала она пытается найти эту переменную в своей области видимости, и только если не находит - обращается в ту область видимости, в которой она была объявлена.

В свою очередь, если и там эта переменная не была найдена, то поиск продолжится в более высокой области видимости и так далее:

```
var b = 10;

function func(a) {
  var c = 100;

  function func2() {
    return a + b + c;
  }

  return func2();
}

func(1);
```

Объявлена функция *func*, а внутри нее еще одна функция - *func2*. *func2* складывает значения переменных, в *func* возвращает результат вызова функции *func2*.

Интерес здесь представляет то, как *func2* будет искать переменные.

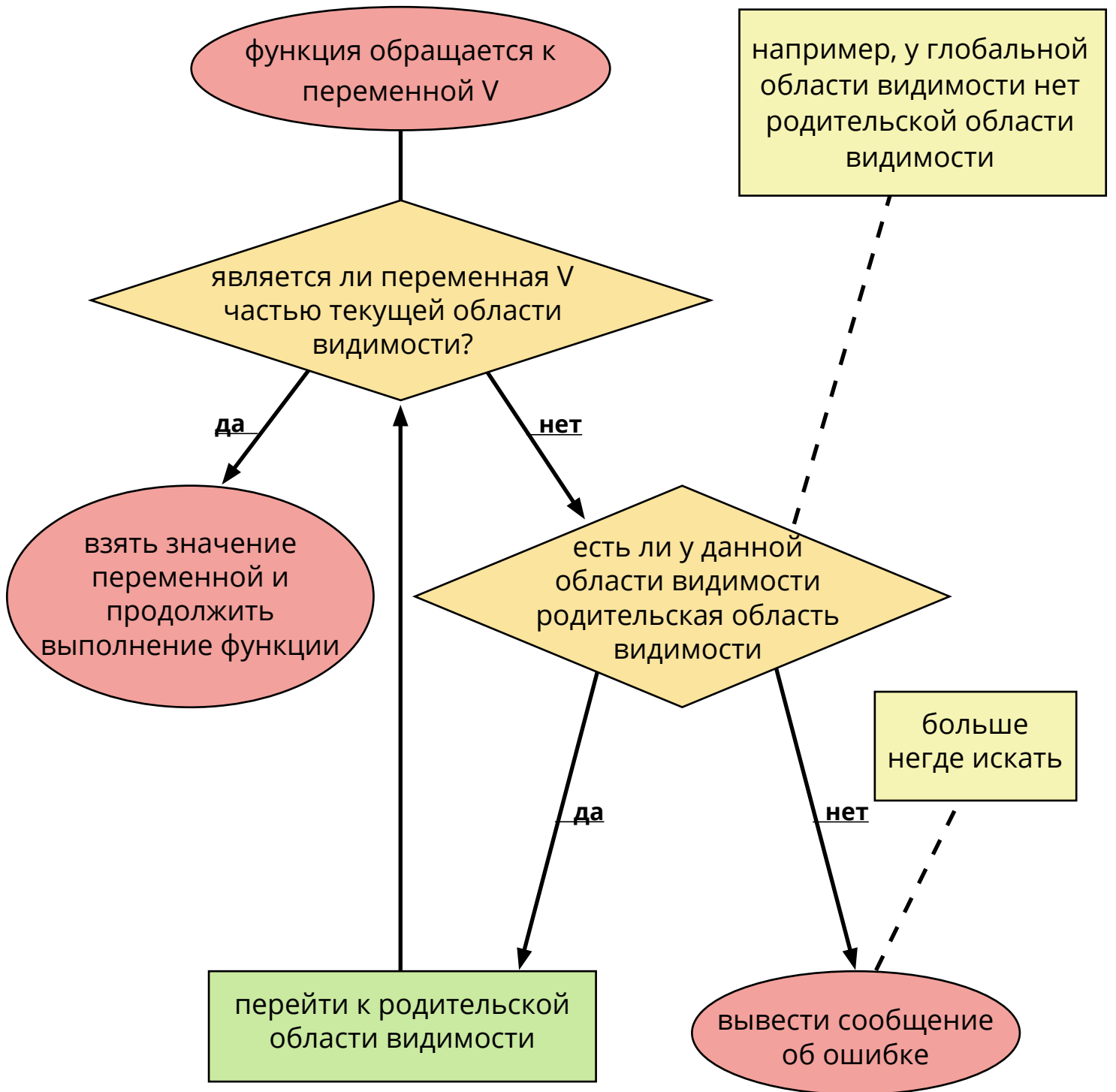
При попытке обратиться к переменной **a** - будет произведен поиск внутри *func2*. Внутри *func2* нет переменной **a**, значит, как говорилось выше - поиск продолжится в той области видимости, в которой была объявлена *func2*. В данном случае, этой областью видимости является область видимости функции *func*. И уже внутри *func* будет найдена переменная **a**.

Поиск переменной **b** будет произведен по тому же алгоритму.

Разница только в том, что внутри *func* нет переменной **b**, а значит поиск продолжится в области видимости, которая находится уровнем выше - в глобальной области, в которой и будет найдена переменная **b**.

С переменной **c** дело будет обстоять так же как и с переменной **b**, т.к. переменная **c** является частью области видимости *func*.

Указанные ниже блок-схема отражает алгоритм поиска переменных внутри функций:



5

Анонимные функции

При использовании **function expression** задавать имя функции не обязательно. Это может быть в тех случаях, когда мы попросту не используем имя функции. Например - передача одной функции в качестве аргумента в другую:

```
function callFunction(fn) {
  var r = fn();

  console.log('Результат работы функции:', r);
}

callFunction(function() {
  return 10 + 10;
});
```

Объявлена функция *callFunction*, которая имеет параметр с именем **fn**. Задача *callFunction* - принять функцию в качестве аргумента, выполнить эту функцию и вывести результат ее выполнения в консоль.

Как видно из кода, при вызове функции *callFunction* мы просто передаем другую функцию, объявленную при помощи **function expression**.

Ранее говорилось, что, если функция объявлена как часть другого выражения, то считается, что такая функция объявлена при помощи **function expression**.

Но интерес здесь представляет то, что функция, передаваемая в *callFunction* в качестве аргумента является анонимной, то есть не имеет имени. Да, внутри *callFunction* она будет доступна по имени параметра **fn**, но на момент передачи, функция не имеет имени.

6

Возврат функций

Кстати, еще одна интересная особенность функции заключается в том, что она может не только принимать другую функцию в качестве параметра, но и возвращать функцию при помощи все того же оператора `return`:

```
function func1() {  
    var a = 10;  
  
    return function() {  
        return a + 100;  
    }  
}  
  
var f = func1();  
  
console.log(f()) //выведется 110
```

Здесь объявлена функция *func1*, внутри которой объявлена переменная *a*. Соответственно, переменная *a* является частью области видимости функции *func1*.

Далее, *func1* возвращает другую функцию, задача которой - сложить значение переменной *a* со значением 100.

За пределами *func1* создается переменная **f**, в которую присваивается результат вызова функции *func1*.

Напомню, что результатом функции *func1* является другая функция, задача которой - сложить значение переменной *a* со значением 100.

После объявления переменной *f* и присвоения ей результата вызова *func1*, происходит вызов функции **f** и вывод результат на экран.

Посмотрите внимательно на код и на результат.

Исходя из результата вызова функции **f** можно сделать вывод, что когда функция возвращает другую функцию, то это другая функция возвращается запомнив ту область видимости, в которой она была объявлена.

В данном случае, внутри *func1* была объявлена переменная **a**. *func1* возвращает анонимную функцию, которая в своем коде использует переменную **a**, доступную ей через механизм замыканий.

Соответственно, когда *func1* возвращает анонимную функцию, анонимная функция не забывает о своей области видимости и может продолжать ей пользоваться не смотря на то что, выход из *func1* уже произошел.

7

IIIFE

Immediately-invoked function expression - это тип вызова функции при котором функция вызывается сразу же после объявления.

Для этого используется такой шаблон: **(function(*параметры*) {код функции})(*аргументы*);**

Посмотри на пример:

```
(function(a, b) {  
    console.log(a + b);  
})(1, 1);
```

В данном случае объявлена анонимная функция и сразу же вызвана с аргументами **1** и **1**.

8

Всплытие

Всплывание или **Hoisting** - это способность интерпретатора знать о функциях или переменных еще до того, как они будут объявлены.

Посмотрим пример:

```
function sum() {  
    return a + b;  
}  
  
var a = 10,  
    b = 10;  
  
sum(); //вернет 20
```

Объявляется функция, которая использует переменные **a** и **b** еще до того, как они были объявлены.

Чтобы понять почему такой код работает, можно представить себе, что перед выполнением кода, интерпретатор сканирует код на предмет переменных и перемещает их объявление в начало их областей видимости. В результате, интерпретатор "видит" вышеуказанный так:

var a, b;

```
function sum() {  
    return a + b;  
}  
  
a = 10;  
b = 10;  
  
sum(); //вернет 20
```

Теперь в коде нет ничего необычного и всё кажется более логичным: сначала объявляются переменные, затем им присваиваются значения и только после этого происходит вызов функции.

Обратите внимание, что “всплывают” только объявления переменных, но не присваивание им значений!

Посмотрите пример:

```
function sum() {  
    return a + b;  
}  
  
sum(); //вернет NaN  
  
var a = 10,  
    b = 10;
```

Здесь, *sum* была вызвана до того, как переменным **a** и **b** были присвоены значения. А пока переменной не присвоено значение, ее значение равно **undefined**.

A undefined + undefined = NaN.

Вот так интерпретатор “увидит” данный код:

```
var a, b;

function sum() {
  return a + b;
}

sum(); //вернет NaN

a = 10;
b = 10;
```

На самом деле, всплытие - это побочный эффект от замыканий. Вспомним, что замыкание - это способность функции запоминать область видимости, в которой эта функция была объявлена.

Суть в том, что для замыкания не важно - выше или ниже по коду была объявлена переменная.

Это то, что касается всплытия переменных.

Теперь поговорим о том, что касается всплытия функций.

Посмотрим пример:

```
var a = 10,  
    b = 10;  
  
sum(); //вернет 20  
  
function sum() {  
    return a + b;  
}
```

Обратите внимание, что функция объявлена после того, как был произведен ее вызов, но не смотря на это - код работает корректно.

Вспомним, что, перед выполнением кода, интерпретатор перемещает переменные в начало их областей видимости. Это же касается и функций. Вот так интерпретатор будет “видеть” приведенный выше пример:

```
function sum() {  
    return a + b;  
}  
  
var a = 10,  
    b = 10;  
  
sum(); //вернет 20
```

И опять же - теперь всё кажется более логичным.

Всплытие и function expression

Теперь, как и обещал, расскажу об основном отличии между *function expression* и *function declaration*.

Вспомним, что *function declaration* - это функция, объявленная в контексте другого выражения. Давайте вернемся к примеру про всплытие функций:

```
var a = 10,  
    b = 10;  
  
sum(); //вернет 20  
  
function sum() {  
    return a + b;  
}
```

А теперь изменим код так, чтобы *sum* была объявлена как **function expression**:

```
var a = 10,  
    b = 10;  
  
sum(); //ошибка!  
  
var sum = function() {  
    return a + b;  
};
```

Такой код работать не будет, т.к. теперь *sum* является обычной переменной.

Вспомните, что при всплытии переменных, всплывает только их объявление, но не присвоение значений! И вот как интерпретатор “увидит” этот код:

```
var a = 10,  
    b = 10,  
    sum;  
  
sum(); //ошибка!  
  
sum = function() {  
    return a + b;  
}
```

То есть, на момент вызова *sum*, внутри соответствующей переменной еще нет кода функции.

Следовательно - функции, объявленные через **function expression** всплывают как переменные, а не как функции.