Глава 4

Функции

В программировании, согласно принципу DRY (Don't Repeat Yourself), принято выделять повторяющиеся блоки кода в функции и давать им осмысленные имена. Это позволяет поддерживать читаемость кода, а также упростить жизнь программиста: при изменении логики программы или исправлении бага, достаточно внести исправление только в одном месте кода.

Кроме этого, функции позволяют организовать рекурсивный вызов. Также в JavaScript функции выполняют особую роль — они создают области видимости переменных.

4.1. Работа с аргументами функции

4.1.1. Аргументы функции

Покажем, как работать с аргументами функций в JavaScript, на примере функции для нахождения минимума двух чисел:

```
function min(a, b) {
    return a < b ? a : b;
}
```

Эта функция принимает два аргумента и работает следующим образом:

```
min(2, 7);    // 2
min(3, 4, 2); // 3
min(13);      // undefined
```

Если вызвать эту функцию от трех аргументов, ошибки не произойдет, в отличие от многих других языков программирования. При этом последний аргумент в этом случае будет просто проигнорирован.

Если же вызвать эту функцию с одним аргументом, происходит следующее. Значение `b` оказывается не определено (undefined). При сравнении

любого числа и undefined результат всегда будет `false`, поэтому в результате выполения функции будет возвращено значение `b`, то есть `undefined`.

Сделать так, чтобы можно было вызывать функцию от одного аргумента (и функция возвращала этот аргумент), можно банально инвертируя условие в условном операторе:

```
function min(a, b) {
   return a > b ? b : a;
}
```

В таком случае функция по прежнему будет возвращать минимум, если ей переданы два аргумента, и возвращать значение единственного аргумента, если передан только один:

```
min(2, 7);    // 2
min(13);      // 13
```

Однако так сделать получается далеко не всегда. В общем случае можно проверить явно, передан аргумент или нет. Для этого достаточно сравнить этот аргумент и undefined.

```
function min(a, b) {
    if (b === undefined) {
        return a;
    }

    return a < b ? a : b;
}
```

В данном случае, если при явной проверке оказывается, что аргумент b не был передан, функция возвращает значение `a`. Иначе — используется старая логика.

```
min(2, 7);    // 2
min(13);      // 13
```

Такой подход также работает.

4.1.2. Значения по умолчанию

При помощи оператора ИЛИ можно задать значение по умолчанию для аргумента. Такой подход позволяет компактнее реализовать функцию из предыдущего примера:

```
function min(a, b) {
    b = b || Infinity;

    return a < b ? a : b;
}
```

Если `b` передан, то в качестве `b` будет использоваться переданное значение. Если же нет, то `b` будет равен undefined и значение `b || Infinity` вернет бесконечность.

```
min(2, 7);    // 2
min(13);      // 13
```

Такое использование оператора ИЛИ опасно. Например, в следующем примере рассматривается функция для расчета стоимости товара, которая принимает два параметра — цену и количество единиц товара. Если второй параметр не задан, следует считать, что количество единиц товара равно одному.

```
function getCartSum(price, count) {
    count = count || 1;

    return price * count;
}
```

Функция работает вполне ожидаемым образом:

```
getCartSum(27.70, 10); //  277
getCartSum(49.90);     //  49.9
```

Но в случае, когда количество товара 0, функция ведет себя не так, как надо:

```
getCartSum(99999, 0);  //  99999 ???
```

Дело в том, что `0` неявно приводится к `false`, поэтому значение `0 || 1` будет `1`.

Чтобы пример работал правильно, значение по умолчанию для параметра следует устанавливать после явного сравнения с undefined:

```
function getCartSum(price, count) {
    if (count === undefined) {
        count = 1;
    }

    return price * count;
}
```

Теперь функция работает корректно во всех случаях:

```
getCartSum(27.70, 10); //  277
getCartSum(49.90);     //  49.9
getCartSum(99999, 0);  //  0
```

4.1.3. Именованные аргументы

Еще один способ работы с аргументами — использование именованных аргументов. Пусть дана функция для вычисления индекса массы тела:

```
function BMI(params) {
    var height = params.height;

    return params.weight / (height * height);
}
```

Эта функция, строго говоря, принимает один аргумент, а нужные значения извлекаются из переданного в качестве этого аргумента объекта. При вычислении индекса массы тела необходимо знать рост человека и его вес, поэтому вызов этой функции будет выглядеть следующим образом:

```
BMI({ weight: 60, height: 1.7 }) // 20.7
```

Такой подход обладает рядом преимуществ:

- Подходит для случаев, когда есть несколько необязательных аргументов
- Не важен порядок аргументов
- Неограниченное число аргументов
- Легко рефакторить код: можно легко добавить или удалить аргумент из цепочки вызовов.

Также есть несколько недостатков:

- Неявный интерфейс: не читая код функции, невозможно понять, какие аргументы нужно передать, чтобы она отработала правильно.
- Неудобно работать с аргументами внутри самой функции, потому что к ним приходится получать доступ через передаваемый объект.

4.1.4. arguments

Еще один способ работы с аргументами функции заключается в использовании объекта `arguments`.

Согласно статье Arguments object с MDN, объект arguments - это подобный массиву объект, который содержит аргументы, переданные в функцию. У этого объекта есть свойство length, которое содержит количество переданных аргументов, а также можно обратиться по индексу к каждому конкретному аргументу:

```
function example() {
    arguments[1]; // 12
    arguments.length; // 2
}

example(3, 12);
```

В следующем примере определена функция sum, которая складывает два числа. Для работы с аргументами используется объект arguments и оператор ИЛИ.

```
function sum() {
    var a = arguments[0] || 0;
    var b = arguments[1] || 0;

    return a + b;
}
```

Если значение аргумента не было передано, для него, в результате использования оператора ИЛИ, устанавливается значение по умолчанию, равное нулю.

```
sum(3, 12);     // 15
sum(45);        // 45
sum(2, 4, 8);   // 6
```

Если же функция была вызвана от трех аргументов, последний аргумент игнорируется и возвращается сумма первых двух.

Чтобы подсчитать сумму всех переданных чисел, следует использовать свойство length объекта arguments для реализации следующего цикла:

```
function sum() {
    var sum = 0;

    for(var i = 0; i < arguments.length; i++) {
```

```
        sum += arguments[i];
    }

    return sum;
}
```

В результате функция будет работать в случае произвольного числа аргументов:

```
sum(2, 4, 8);   // 14
```

Объект `arguments` — не массив, но может быть приведен к массиву с помощью метода `slice`, заимствованного у массива. Метод `call` позволяет вызвать заимствованный метод от лица объекта `arguments`.

```
function sum() {
    var args = [].slice.call(arguments);

    return args.reduce(function (sum, item) {
        return sum + item;
    });
}
```

Переменная args, таким образом, будет содержать массив, у которого будет доступен метод `reduce`. Функция также будет работать правильно:

```
sum(2, 4, 8);   // 14
```

Метод Call

Разберем подробнее работу метода call. Например, если вызвать метод slice от массива, будет создана копия исходного массива. Кроме прямого вызова, для вызова метода slice можно использовать метод call, передавая исходный массив в качестве единственного аргумента. В этом случае также будет создана копия исходного массива.

```
function example() {
    [1, 2].slice();           // [1, 2]
    [].slice.call([3, 4]);    // [3, 4]

    [].slice.call(arguments); // [5, 6]
}

example(5, 6);
```

Кроме массивов метод slice может принимать массивоподобные объекты, каким и является объект arguments.

4.2. Объявление функции

4.2.1. function declaration

Существует несколько способов объявления функции. Первый способ уже встречался ранее и называется function declaration.

```
// function declaration
function add(a, b) {
    return a + b;
}
```

В этом случае после ключевого слова function указывается ее имя.

4.2.2. function expression

Существует и альтернативный способ, который называется function expression:

```
// function expression
var add = function (a, b) {
    return a + b;
}
```

В этом способе значение функции присваивается некоторой переменной.

4.2.3. Отличия

Оба этих способа можно использовать, но следует учитывать, что их поведение несколько отличается. В случае использования function declaration вызов функции может быть до момента ее объявления:

```
add(2, 3);  // 5

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

В случае же function expression это приводит к ошибке:

```
add(2, 3);  // TypeError

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

4.2.4. Named function expression

Можно объединить эти два способа и получить третий способ объявления функции — named function expression.

```
var factorial = function inner(n) {
    return n === 1 ?
        1 : n * inner(n - 1);
}
```

В этом случае имя функции, которое указано после ключевого слова function, будет доступно только внутри функции, а по имени переменной, в которую присваивается значение функции, функция будет доступна только снаружи:

```
typeof factorial; // 'function'
typeof inner;     // ReferenceError
```

Имя функции inner не доступно снаружи функции и попытка определения ее типа приводит к ошибке интерпретатора. Но она доступна внутри самой функции, в чем можно убедиться непосредственно вызывав ее:

```
factorial(3);    // 6
```

4.2.5. Конструктор Function

Существует еще один, достаточно экзотический, способ объявления функции с помощью конструктора Function:

```
var add = new Function('a', 'b', 'return a + b');
```

Сначала перечисляются через запятую как строки имена аргументов функции, а последним аргументом — тело функции, также в виде строки.

Созданную таким образом функции также без проблем можно использовать:

```
add(2, 3);  // 5
```

Применяется такой способ редко, обычно когда код функции генерируется «на лету», то есть когда код функции генерируется другим кодом. Использовать такой способ объявления, вообще говоря, нежелательно. Во-первых, становится сложно ориентироваться в коде. Во-вторых, интерпретатор не сможет оптимизировать код созданной так функции, а значит функция может работать значетельно медленнее, чем в случае, если бы она была объявлена иным способом.

4.3. Область видимости

4.3.1. Глобальный объект

Любая переменная или функция, которая была объявлена не в теле другой функции, объявлена в глобальной области видимости.

```
var text = 'Привет'; // ┌{ text, greet }
                     // │
function greet() {   // │
}                    // │
```

Переменная text и функция greet оказываются объявленными в глобальной области видимости и доступны на протяжении всего кода. Получить значение переменной text можно через свойство text объекта global:

```
global.text; // 'Привет'
```

4.3.2. Создание области видимости

Новую область видимости можно создать при помощи функции. При объявлении функции greet создается новая область видимости, в которую помещаются все аргументы функции и объявленные в ней переменные.

```
function greet() {             // ┌{ greet }
    var text = 'Привет';       // ├─┌{ text }
    text; // 'Привет'          // │ │
}                              // │ │
                               // │
text;   // ReferenceError:     // │
        //    text is not defined
```

В этом примере в глобальной области видимости оказывается функция greet, а в области видимости функции — переменная text. Переменная text перестает быть доступной после того, как функция завершает свое выполнение.

4.3.3. Нет блочной области видимости

Согласно ECMAScript 5.1, область видимости создается только функцией. То есть переменные, которые были объявлены в блоке кода, например внутри условного оператора, будут доступны за пределами этого блока.

```
function greet() {              // ┌{ greet }
    if (true) {                 // ├──{ text }
        var text = 'Привет';    //
    }                           //
                                //
    text; // 'Привет'           //
}                               //
```

4.3.4. Вложенные функции

Функции, объявленные внутри других функций, называются вложенными функциями.

```
function greet() {              // ┌{ greet }
    var text = 'Привет';        // ├──{ text, nested }
                                //
    function nested() {         //    ├──{ }
        text; // 'Привет'       //
    }                           //
}                               //
```

При этом переменные из области видимости родительской функции становятся доступными дочерней.

4.3.5. Затенение

Если в родительской функции и в дочерней объявить переменные с одинаковыми именами, будет иметь место затенение.

```
function greet() {              // ┌{ greet }
    var text = 'Привет';        // ├──{ text: Привет, nested }
                                //
    function nested() {         //    ├──{ text: Пока }
        var text = 'Пока';      //
        text; // 'Пока'         //
    }                           //
                                //
    text; // 'Привет'           //
}                               //
```

В области видимости дочерней функции по имени text доступна переменная со значением «Пока», а в области видимости родителькой — переменная со значением «Привет».

4.4. Всплытие

При обращении к переменной до момента ее объявления не возникает ошибки. Механизм, позволяющий это, называется всплытие.

Выполнение кода можно условно разделить на две части:

- Инициализация: Интерпретатор просматривает весь код на предмет объявления функций и переменных:
  - function declaration
  - var

  Это и называется всплытием.
- Собственно выполнение

Фукнции и переменные «всплывают» немного по-разному.

```
add(2, 3);                  // { add: function }

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

Если функция add была объявлена через function declaration, то после инициализации в add будет лежать функция, которая может быть вызвана до момента своего объявления.

```
add(2, 3);  // 5

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

Если же функция была объявлена через function expression, до момента объявления в add будет находиться значение undefined.

```
add(2, 3);                  // { add: undefined }

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

Использовать функцию до момента ее объявления не получится, поскольку в этом случае операция «круглые скобки» применяется к undefined, что приводит к TypeError:

```
add(2, 3); // TypeError

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

В переменной add функция окажется только в момент присваивания.

```
add(2, 3); // TypeError

var add = function (a, b) {
    return a + b;          // { add: function }
}
```

Всплытие переменных работает в пределах области видимости. Это проиллюстрировано на следующем примере:

```
                           // { greet: function }
function greet(){          // { greet: function, text:
 ↪  undefined  }
  var text = "Привет";     // { greet: function, text:
   ↪  "Привет"  }
}                          // { greet: function }
                           // { greet: function }
```

В глобальной области видимости доступна только функция greet. Внутри функции всплывает переменная text, которая до момента присваивания имеет значение undefined. После завершения функции переменная text перестает быть доступной.

4.4. Замыкание

Замыкание — это функция со всеми ее внешними переменными, к которым она имеет доступ.
При объявлении новых переменных выделяется новый участок памяти. Все переменные, на которые никто не ссылается, сборщик мусора JavaScript может вычистить с помощью счетчика ссылок.
Внутри функции makeCounter мы имеем доступ к переменной currentCount, поэтому счетчик ссылок на эту переменную не ноль. Однако и за пределами этой функции мы по-прежнему продолжаем ссылаться на эту переменную. Мы делаем это неявно через функцию Counter, которая имеет доступ к currentCount. В этом и есть смысл замыкания.

```
function makeCounter() {
    var currentCount = 0; // { currentCount: 1 }
    return function () {
        return currentCount++;
    };
}

var counter = makeCounter(); // { currentCount: 1 }
```

makeCounter в качестве результата возвращает новую функцию. В JavaScript область видимости создается функциями. При этом внутри этой функции мы обращаемся к переменной currentCount.
Ищем currentCount в области видимости функции, которую возвращаем. Не находим ничего и идем в родительскую функцию. Там объявлена переменная currentCount, к ней мы имеем доступ. Таким образом, вызывая функцию Counter, мы будем увеличивать счетчик внешней переменной на 1.

```
function makeCounter() {
    var currentCount = 0;
    return function () {
        return currentCount++;
    };
}
```

```
// ┌{ makeCounter } // 1
// ├─┬{ currentCount } // 2
// │ │
// │ │
// │ ├─┬{ } // 3
// │ │ │
// │ │
// │
```

Однако если мы позовем функцию makeCounter еще раз, мы получим новую функцию, которая ссылается уже на новую переменную currentCount вне зависимости от первой переменной.

```
var counter = makeCounter();
counter(); // 0
counter(); // 1
counter(); // 2

var yetAnother = makeCounter();
yetAnother(); // 0
```

Объявим функцию greet, которая принимает переменную name и возвращает новую функцию. Поскольку переменная name является внешней по отношению к этой функции, имеет место замыкание. Вызывая функцию greet с некоторой строкой, например, строкой ('мир!'), мы получаем новую функцию helloWorld, которая при вызове возвращает строку "Привет, мир!"

```
function greet(name) {
    return function () {
        return 'Привет,' + name;
    }
}

var helloWorld = greet('мир!');
helloWorld(); // "Привет, мир!"
```

4.4.1. Модуль

Определим в нашем коде две функции. Функцию format, которая преобразует переданную дату в строку. И функцию getDateString, которая определяет Date, если не определено текущим временем, и возвращает время в виде строки. Если мы вызовем getDateString, не передав туда аргументы, мы получим текущую дату в виде строчки.

```
function format(date) {
    return date.ToGMSstring()
}

function getDateString(date){
    date = date || new Date();
    return format(date);
}
```

Если кто-то случйно переопределит функцию format, функция getDateString будет испорчена. Вместо того, чтобы возвращать текущую дату в виде строки, она будет возвращать новую строку, определенную в новой функции format.
Чтобы защитить поведение функции getDateString, мы сможем воспользоваться pattern module. Для того чтобы определить модуль, мы воспользуемся самовызывающейся функцией. Функцию format и getDateString мы обернем в новую функцию, которую сразу же и вызовем. Из этой функции мы будем возвращать функцию getDateString и складывать в переменную с одноименным названием. Для того чтобы показать, что это самовызывающаяся функция, до ключевого слова function ставится открывающаяся скобка.

```
var getDateString = (function () {
    function format(date) {
        return date.toGMTString()
}

return function getDateString(date) {
    date = date || new Date();
    return format(date);
    }
}());
```

Функции format и getDateString находятся внутри новой самовызывающейся функции: портить их значения нельзя.
Для того чтобы написать самовызывающуюся функцию, мы должны определить ее в режим function expression: чтобы интерпретатор отличил это от function declaration, мы ставим открывающуюся круглую скобочку до ключевого слова function. Закрывающуюся круглую скобочку мы можем поставить как после вызова самовызывающейся функции, так и до ее вызова.

```
(function() {
}());

(function() {
}());
```

С использованием самовызывающихся функций мы можем реализовать pattern module.

- Область видимости в JavaScript ...
- Замыкания в JavaScript
- Замыкания, область видимости
- Лексическая область видимости