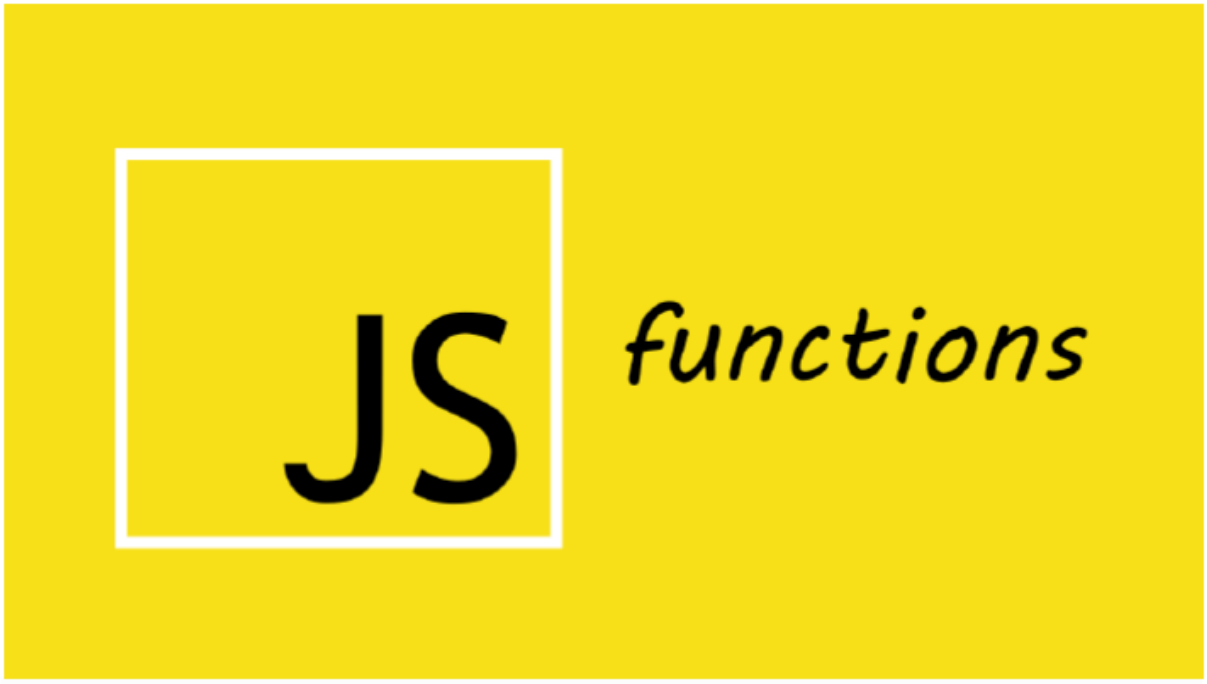
Лабораторная работа №9. Функции в JavaScript и классический способ их создания



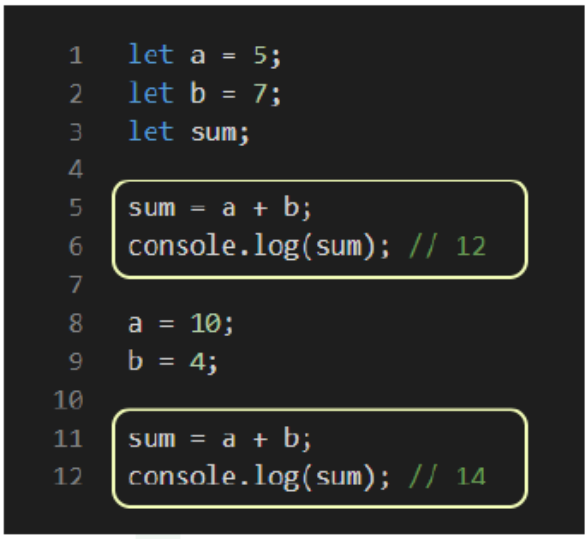
Содержание:

На этом занятии рассмотрим, что такое функция и зачем она нужна. Разберём классический способ её объявления, параметры, аргументы и оператор return.

Что такое функция?

Функция – это фрагмент кода, который можно выполнить многократно в разных частях программы. Т.е. одни и те же действия много раз с разными исходными значениями.

В следующем примере имеются повторяющиеся блоки кода, которые можно вынести отдельно в JavaScript функцию:



```
let a = 5;
let b = 7;
let sum;

sum = a + b;
console.log(sum); // 12

a = 10;
b = 4;

sum = a + b;
console.log(sum); // 14
```

Функция `sum`, в которую вынесен повторяющийся блок кода. После этого в местах программы, в которых его нужно выполнять, просто помещён вызов этой функции:

```
let a = 5;
let b = 7;

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

sum(a, b); // 12

a = 10;
b = 4;

sum(a, b); // 14
```

Это один из классических сценариев использования функций, который позволяет значительно упростить написание программ на JavaScript.

Кроме этого, функции позволяют структурировать код. Например, если перед вами стоит какая-то задача, то чтобы её проще написать, её можно разбить на подзадачи и оформить каждую из них в виде функции. А затем уже используя их написать финальный код. Вдобавок к этому в такой код будет проще вносить различные изменения и добавлять новые возможности.

JavaScript позволяет создавать функцию различными способами:

- **Function Declaration**;
- **Function Expression**;
- **Arrow Function**.

В этой статье разберём первый классический способ, который называется **Function Declaration**.

Объявление и вызов функции

Операции с функцией в JavaScript можно разделить на 2 этапа:

- **объявление (создание) функции;**
- **вызов (выполнение) этой функции.**

1. Объявление функции. Написание функции посредством **Function Declaration** начинается с ключевого слова `function`. После чего указывается **имя**, **круглые скобки**, внутрь которых при необходимости помещаются параметры, и **тело**, заключённое в фигурные скобки. Имя функции ещё очень часто называют названием. В теле пишутся те действия, которые вы хотите выполнить с помощью этой функции.

```
// nameFn – имя функции, params – параметры
function nameFn (params) {
  // тело функции
}
```

Например:

```
// объявляем функцию someName
function someName() {
  console.log('Вы вызвали функцию someName!');
}
// function – ключевое слово, которое означает, что мы создаём функцию
// someName – имя функции
// () – круглые скобки, внутри которых при необходимости описываются параметры
// { ... } – тело функции
```

При составлении имени функции необходимо руководствоваться такими же правилами, что для переменных. Т.е. можно использовать буквы, цифры (0 – 9), знаки «$» и «_». В качестве букв рекомендуется использовать английский алфавит (a-z, A-Z). Имя функции, также как и имя переменной не может начинаться с цифры.

При этом функции рекомендуется давать осмысленное название, т.е. такое, что было понятно, что она делает только исходя из её имени. Кроме этого, не нужно делать так, чтобы одна функция выполняла разные задачи. Лучше создать много различных функций, каждая из которых будет выполнять одну строго определённую задачу.

Параметры предназначены для получения значений, переданных в функцию, по имени. Их именование осуществляется также как переменных:

```
// объявляем функцию с двумя параметрами
  function fullname(firstname, lastname) {
  console.log(`${firstname} ${lastname}`);
}
```

Параметры ведут себя как переменные и в теле функции мы имеем доступ к ним. Значения этих переменных (в данном случае `firstname` и `lastname`) определяются в момент вызова функции. Обратиться к ним вне функции нельзя.

Если параметры не нужны, то круглые скобки всё равно указываются.

2. Вызов функции. Объявленная функция сама по себе не выполняется. Запуск функции выполняется посредством её вызова.

При этом когда мы объявляем функцию с именем, мы тем самым по сути создаём новую переменную с этим названием. Эта переменная будет функцией. Для вызова функции необходимо указать её имя и две круглые скобки, в которых при необходимости ей можно передать аргументы. Отделение одного аргумента от другого выполняется с помощью запятой.

```
// объявляем функцию someName
function someName() {
  console.log('Вы вызвали функцию someName!');
}
// вызываем функцию someName
someName();
```

Параметры и аргументы

Параметры, как мы уже отметили выше – это по сути переменные, которые описываются в круглых скобках на этапе объявления функции. Параметры доступны только внутри функции, получить доступ к ним снаружи нельзя. Значения параметры получают в момент вызова функции, т.е. посредством аргументов.

Аргументы – это значения, которые мы передаём в функцию в момент её вызова.

```
// userFirstName и userLastName – параметры (userFirstName будет иметь значение
первого аргумента, а userLastName соответственно второго в момент вызова этой
функции)
function sayWelcome (userFirstName, userLastName) {
  console.log(`Добро пожаловать, ${userLastName} ${userFirstName}`);
}
// 'Иван' и 'Иванов' – аргументы
sayWelcome('Иван', 'Иванов'); // Добро пожаловать, Иванов Иван
// 'Петр' и 'Петров' – аргументы
sayWelcome('Петр', 'Петров'); // Добро пожаловать, Петров Петр
```

При вызове функции в JavaScript **количество аргументов не обязательно должно совпадать с количеством параметров**. Если аргумент не передан, а мы хотим его получить с помощью параметра, то он будет иметь значение `undefined`.

```
function sayWelcome (userFirstName, userLastName) {
  console.log( `Добро пожаловать, ${userLastName} ${userFirstName} `);
}
// с одним аргументом
sayWelcome('Иван'); // Добро пожаловать, undefined Иван
// без передачи аргументов
sayWelcome(); // Добро пожаловать, undefined undefined
```

Передача аргументов примитивных типов осуществляется **по значению**. Т.е. значение переменной не изменится снаружи, если мы изменим её значение внутри функции.

```
let a = 7;
let b = 5;

function sum (a, b) {
  a *= 2; // 14
  console.log(a + b);
}
sum(a, b); // 19
console.log(a); // 7
```

Передача значения по ссылке

Очень важно отметить при работе с функциями, что когда в качестве аргумента мы указываем ссылочный тип, то его изменение внутри функции изменит его и снаружи:

```
// объявим переменную someUser и присвоим ей объект, состоящий из двух свойств
const someUser = {
  firstname: 'Петр',
  lastname: 'Петров'
}
// объявим функцию changeUserName
function changeUserName(user) {
  // изменим значение свойства firstname на новое
  user.firstname = 'Александр';
}
// вызовем функцию changeUserName
changeUserName(someUser);
// выводим значение свойства firstname в консоль
console.log(someUser.firstname); // Александр
```

В этом примере переменные `someUser` и `user` ссылаются на один и тот же объект в памяти. И когда мы изменяем объект внутри функции, то `someUser` тоже изменится.

При этом не рекомендуется изменять внешние относительно функции переменные. Т.е. если на вход функции мы передаём объект и хотим изменить его, то лучше создать копию этого объекта и изменять уже его. Таким образом мы не изменим внешний объект и код функции останется чистой.

Чтобы избежать изменение внешнего объекта, который мы передаем в функцию через аргумент, необходимо создать копию этого объекта, например, посредством `Object.assign`:

```
const someUser = {
  firstname: 'Петр',
  lastname: 'Петров'
}
function changeUserName(user) {
  // создадим копию объекта user
  const copyUser = Object.assign({}, user);
  // изменим значение свойства firstname на новое
  copyUser.firstname = 'Александр';
  // вернём копию объекта в качестве результата
  return copyUser;
}

// вызовем функцию и сохраним результат вызова функции в переменную updatedUser
const updatedUser = changeUserName(someUser);
// выводим значение свойства firstname в консоль для объекта someUser
console.log(someUser.firstname); // Петр
// выводим значение свойства firstname в консоль для объекта updatedUser
console.log(updatedUser.firstname); // Александр
```

В этом примере мы внутри функции создали новый объект `copyUser`, который является копией объекта, переданного функции в качестве аргумента в момент её вызова. Т.е. `someUser` и `copyUser` - это разные объекты, хоть и содержащие на этапе копирования одинаковые свойства. После копирования, мы уже меняем свойство нового объекта и возвращаем его в качестве результата выполнения функции.

На практике бывают ситуации, когда внутри функции происходит изменение внешних объектов. Но в большинстве случаев изменять оригинальный объект или какие-то другие внешние переменные внутри функции не рекомендуется.

Локальные и внешние переменные

Переменные, объявленные внутри функции, называются локальными. Они не доступны вне функции. По сути это переменные, которые действуют только внутри функции.

```
let a = 7;
// объявление функции sum
function sum(a) {
  // локальная переменная функции
  let b = 8;
  console.log(a + b);
}
// вызов функции sum
sum(a);
console.log(b); // Error: b is not defined
```

При этом, когда мы обращаемся к переменной и её нет внутри функции, она берётся снаружи. Переменные объявленные вне функции являются по отношению к ней **внешними**.

```
// внешние переменные
let a = 7;
let b = 3;
function sum() {
```

```
  // локальная переменная
  let a = 8;
  // изменение значения внешней переменной (т.к. b нет внутри функции)
  b = 4;
  console.log(a + b);
}
sum(); // 12
```

Работа с аргументами через arguments

Получить доступ к аргументам в JavaScript можно не только с помощью параметров, но также посредством специального массивоподобного объекта `arguments`, доступ к которому у нас имеется внутри функции. Кроме этого он также позволяет получить количество переданных аргументов.

```
function myFn() {
  // проверим является ли arguments обычным массивом
  console.log(Array.isArray(arguments)); // false
  // количество аргументов
  console.log(arguments.length); // 0
}

myFn(); // false
```

Доступ к аргументам через `arguments` выполняется точно также как к элементам обычного массива, т.е. по порядковому номеру:

```
// объявление функции sum
function sum() {
  const num1 = arguments[0]; // получаем значение 1 аргумента
  const num2 = arguments[1]; // получаем значение 2 аргумента
  console.log(num1 + num2);
}

sum(7, 4); // 11
```

Получение аргументов через `arguments` в основном используется, когда мы заранее не знаем их точное количество. Например, создадим функцию, которая будет подсчитывать сумму всех числовых аргументов:

```
function sum() {
  let sum = 0;
  // arguments.length - число аргументов
  for (let i = 0, length = arguments.length; i < length; i++) {
    if (typeof arguments[i] === 'number') {
      sum += arguments[i];
    }
  }
  console.log(sum);
}

sum(4, 20, 17, -6); // 35
sum('', 3, -5, 32, null); // 30
```

Через цикл `for...of` этот пример можно записать так:

```
function sum() {
  let sum = 0;
  for (argument of arguments) {
    if (typeof argument === 'number') {
      sum += argument;
    }
  }
  console.log(sum);
}
```

При необходимости `arguments` можно преобразовать в обычный массив:

```
function names() {
  // превращаем arguments в полноценный массив
  const args = Array.from(arguments);
  console.log(args);
}
names('Даша', 'Маша', 'Нина'); // ["Даша", "Маша", "Нина"]
```

В JavaScript `arguments` можно использовать для написания очень гибких функций, которые в зависимости от количества аргументов, и, их типа, могут выполнять одни или другие действия.

Все переменные, созданные внутри функции и её параметры, как мы уже отмечали выше, являются её локальными переменными. Они доступны только внутри неё, а также в других функциях, вложенных в неё, если там нет переменных с такими же именами. Вне функции локальные переменные не доступны:

```
function fnA(a, b) {
  const c = 4;
  function fnB(d) {
    console.log(a + b + c + d);
  }
  fnB(1);
}
fnA(3, 5); // 13
fnA(4, 7); // 16
// переменная a не доступна за пределами функции fnA
console.log(a); // Uncaught ReferenceError: a is not defined
```

При этом внутри функции мы имеем доступ к внешним переменных, но только к тем, которых с таким же именем нет в текущей функции:

```
const a = 10;
function fa(b) {
  console.log(a + b);
}

fa(7); // 17
```

Колбэк функции

Колбэк функция (от английского *callback function*) – это обычная функция, которая просто вызывается внутри другой функции. Такие функции ещё называются функциями обратного вызова. Они очень часто применяются в асинхронном коде.

Передаётся колбэк функция в другую через аргумент:

```
// колбэк функция
function cb() {
  console.log('callback');
}
// функция, которая будет принимать на вход колбэк функцию
function fnWithCb(cbFn) {
  console.log('before calling the callback function');
  cbFn();
}
// вызываем функцию fnWithCb() и передаём ей в качестве аргумента колбэк функцию cb
fnWithCb(cb);
```

В этом примере имеются две функции:

- `cb` – её будем использовать в роли колбэк функции, т.е. вызывать в `fnWithCb`;
- `fnWithCb` – эта функция, которая содержит параметр `cbFn`, он будет при её вызове принимать другую функцию (в данном случае `cb`).

Таким образом, функция `cb` вызывается внутри функции `fnWithCb`. В неё она передаётся как аргумент. Без вызова `fnWithCb` она не вызовется, т.к. она вызывается только внутри `fnWithCb`.

Другой пример:

```
// объявим колбэк функцию
function printToLog(message) {
  console.log(message);
}
// объявим функцию, имеющую 3 параметра
function sum(num1, num2, callback) {
  // вычислим сумму 2 значений и запишем его в result
  const result = num1 + num2;
  // вызовем колбэк функцию
  callback(result);
}
// вызовем функцию sum
sum(5, 11, printToLog);
```

Ещё один пример:

```
// колбэк функция
function setColorBody() {
  document.body.style.backgroundColor = '#00ff00';
```

```
}
// функция, которая будет вызывать функцию setColorBody через 3 секунды
setTimeout(setColorBody, 3000);
```

В этом примере у нас имеется функция `setColorBody`. В теле она содержит код, который устанавливает цвет фона `<body>`.

Далее вызывается функция `setTimeout()`. Она в свою очередь вызывает где-то внутри себя функцию, переданную ей в качестве первого аргумента. Причем вызывает не сразу, а через указанное количество миллисекунд. В данном случае через 3000.

Функцию `setTimeout` мы нигде не объявляли, т.к. она присутствует в JavaScript по умолчанию. Оня является методом глобальное объекта `window` в браузере и `global` в Node.js.

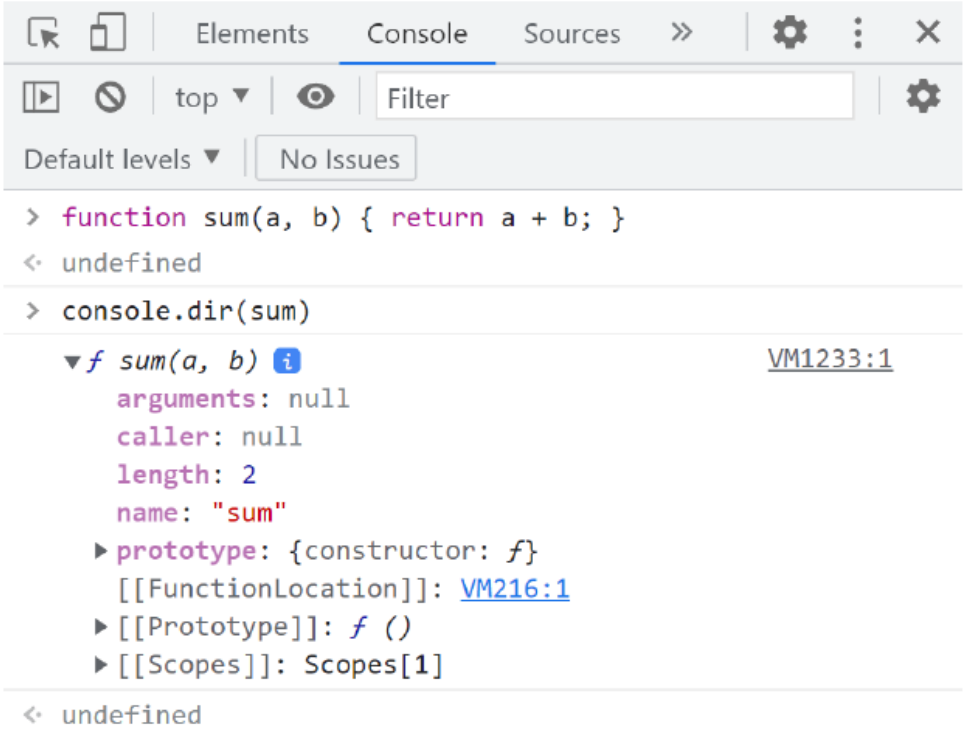
Обратите внимание, что в `setTimeout` мы просто передаём функцию `setColorBody`. Т.е. сами её не вызываем.

Также возможны ситуации, когда одна колбэк функция вызывает другую колбэк функцию.

В JavaScript всё это возможно, благодаря тому, что функции являются объектами. А так как функция является объектом, её как значение можно передавать в другие функции посредством аргументов.

Функция – это объект

Функция в JavaScript, как уже было отмечено выше – это определённый тип объектов, которые можно вызывать. А если функция является объектом, то у неё как у любого объекта имеются свойства. Убедиться в этом очень просто. Для этого можно воспользоваться методом `console.dir()` и передать ему в качестве аргумента функцию.



На изображении показана структура функции `sum`. Используя точечную запись, мы например, можем получить название функции (свойство `name`) и количество параметров(`length`):

```
console.log(sum.name); // sum
console.log(sum.length); // 2
```

Узнать является ли переменная функцией можно с помощью `typeof`:

```
function myFunc() {};

console.log(typeof myFunc); // function
```

Например, проверим является переменная колбэк функцией перед тем её вызвать:

```
function sum(num1, num2, callback) {
  const result = num1 + num2;
  if (typeof callback === 'function') {
    callback(result);
  }
}
```

Возврат значения

Функция всегда возвращает значение, даже если мы не указываем это явно. По умолчанию она возвращает значение `undefined`.

Для явного указания значения, которое должна возвращать функция используется инструкция `return`. При этом значение или выражение, результат которого должна вернуть функция задаётся после этого ключевого слова.

```
// expression - выражение, результат которого будет возвращен функцией myFn
function myFn() {
  return expression;
}
```

Если `return` не указать, то функция всё равно возвратит значение, в данном случае `undefined`.

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

// вызовем функцию и сохраним её результат в константу result
const result = sum(4, 3);
// выведем значение переменной result в консоль
console.log(result); // undefined
```

С использованием инструкции `return`:

```
function sum(a, b) {
  // вернём в качестве результата a + b
  return a + b;
}

// вызовем функцию и сохраним её результат в константу result
const result = sum(4, 3);
// выведем значение переменной result в консоль
console.log(result); // 7
```

Инструкции, расположенные после `return` никогда не выполняются:

```
function sum(a, b) {
  // вернём в качестве результата a + b
  return a + b;
  // код, расположенный после return никогда не выполнится
  console.log('Это сообщение не будет выведено в консоль');
}

sum(4, 90);
```

В этом примере, функция `sum` возвращает число 94 и прекращает выполнение дальнейших инструкций после `return`. А так как работа функции закончилась, то сообщение в консоль выведено не будет.

Функция, которая возвращает функцию

В качестве результата функции мы можем также возвращать функцию.

Например:

```
function outer(a) {
  return function(b) {
    return a * b;
  }
}

// в one будет находиться функция, которую возвращает outer(3)
const one = outer(3);
// в two будет находиться функция, которую возвращает outer(4)
```

```
const two = outer(4);

// выведем в консоль результат вызова функции one(5)
console.log(one(5)); // 15
// выведем в консоль результат вызова функции two(5)
console.log(two(5)); // 20
```

Вызовы функции `outer(3)` и `outer(4)` возвращают одну и туже функцию, но первая запомнила, что `a = 3`, а вторая - что `a = 4`. Это происходит из-за того, что функции в JavaScript «запоминают» окружение, в котором они были созданы. Этот приём довольно часто применяется на практике. Так как с помощью него мы можем, например, на основе одной функции создать другие, которые нужны.

В примере, приведённом выше, мы могли также не создавать дополнительные константы `one` и `two`. А вызвать сразу после вызова первой функции вторую.

```
// выведем в консоль результат вызова функции one(5)
console.log(outer(3)(5)); // 15
// выведем в консоль результат вызова функции two(5)
console.log(outer(4)(5)); // 20
```

При создании таких конструкций нет ограничений по уровню вложенности, но с точки зрения разумности этим лучше не злоупотреблять.

Функцию, приведённую в коде мы можем также создать и так:

```
function outer(a) {
  function inner(b) {
    return a * b;
  }
  return inner;
}
```

Кроме этого в качестве результата мы можем также возвратить внешнюю функцию:

```
function fa() {
  return 'Привет!';
}

function fb() {
  return fa;
}

fb()(); // Привет!
```

Рекурсия

Функцию можно также вызвать внутри самой себя. Это действие в программировании называется рекурсией.

Кроме этого необходимо предусмотреть условия для выхода из рекурсии. Если это не сделать функция будет вызывать сама себя до тех пор, пока не будет брошена ошибка, связанная с переполнением стека.

Например, использование рекурсии для вычисления факториала числа:

```
function fact(n) {
  // условие выхода из рекурсии
  if (n === 1) {
    return 1;
  }

  // возвращаем вызов функции fact(n - 1) умноженное на n
  return fact(n - 1) * n;
}
console.log(fact(5)); // 120
```

Пример, в котором используя рекурсию выведем числа от указанного до 10:

```
function counter(value) {
  // условие выхода из рекурсии
  if (value < 10) {
    console.log(value);

    // возвращаем вызов функции counter(value + 1)
    return counter(value + 1);
  }
}

counter(1); // 1, 2, 3, 4, 5, 6, 7, 8, 9
counter(7); // 7, 8, 9
```

Перегрузка функций в JavaScript

Перегрузка функций в программировании – это возможность объявлять в одном месте несколько функций с одинаковыми именами. Отличаются такие функции друг от друга параметрами. Используется перегрузка функций для того, чтобы можно было вызвать подходящую под переданные аргументы функцию.

В JavaScript не реализована перегрузка функций в том виде, как это реализовано в Си или других языках. Но подобную функциональность можно имитировать в JavaScript. Для этого у нас есть всё, что для этого необходимо.

Например, того чтобы проверить имеет параметр значение или нет, мы можем проверить его значения на `undefined`. Узнать количества переданных аргументов функции можно через `arguments.length`. Определить значения параметра можно используя `typeof` или `instanceof`.

Например, создадим функцию `bodyBgColor`, которая будет иметь 2 режима работы. Если её вызвать без аргументов, то она будет возвращать цвет фона `body`. А если с текстовым аргументом, то она будет устанавливать цвет фона `body`.

```
// объявление функции bodyBgColor
function bodyBgColor(color) {
  // если параметр color имеет в качестве значения строку, то установим цвет фона
body
  if (typeof color === 'string') {
    document.body.style.backgroundColor = color;
  }

  // вернём в качестве результата текущий цвет фона body
  return getComputedStyle(document.body).backgroundColor;
}

// получим текущий цвет body и выведем его в консоль
console.log(bodyBgColor());

// установим новый цвет фона body
bodyBgColor('green');
```

Пример реализации «перегруженной» функции для вычисления оптимального количества ккал, которых необходимо человеку в день:

```
function calculateCalories(gender, height) {
  let result = gender === 'man' ? (height - 100) * 20 : (height - 105) * 19;
  if (typeof arguments[2] === 'number') {
    result *= arguments[2];
  }
  return result.toFixed(0);
}

console.log(`Оптимальное кол-во ккал: ${calculateCalories('man', 185)}`);
console.log(`Оптимальное кол-во ккал: ${calculateCalories('woman', 168, 1.2)}`);
console.log(`Оптимальное кол-во ккал: ${calculateCalories('woman', 168)}`);
```

Значение параметров функции по умолчанию

Параметрам функции можно присвоить дефолтное значение. Оно будет использоваться, когда при вызове функции этому параметру не будет задано значение с помощью аргумента:

```
function setBgColor(selector, color = 'green') {
  const el = document.querySelector(selector);
  el.style.backgroundColor = color;
}
setBgColor('body');
```

При вызове функции с одним аргументом, второму параметру будет автоматически присвоено строка `'green'`.

Работу параметра по умолчанию можно представить так:

```
function setBgColor(selector, color) {
  const el = document.querySelector(selector);
  color = color === undefined ? 'green' : color;
  el.style.backgroundColor = color;
}
setBgColor('body');
```

Пример функции `addNewMessage`, в которой один из параметров имеет дефолтное значение. Причем это значение будет меняться в зависимости от текущей даты:

```
const messages = [];
function addNewMessage(text, date = new Date()) {
  messages.push({
    text,
    date
  })
};
addNewMessage('Как сделать вкусный коктейль в блендере?');
addNewMessage('Какое мороженое лучше для коктейля?');
console.log(messages);
```

В этом примере значение дефолтного параметра `date` будет определяться в момент вызова функции `addNewMessage` исходя из текущей даты. То есть, если вы вызовите `addNewMessage()` в другое время, то `date` будет иметь другую дату.

Остаточные параметры

В вызове функции ей можно передать больше аргументов, чем у неё имеется параметров. Получить все остальные аргументы можно в виде массива с помощью синтаксиса «остаточные параметры»:

Если при вызове функции ей передать аргументов больше, чем у нас есть параметров, то получить оставшиеся значения можно с помощью, так называемых **rest patameters**.

```
// ...nums – остаточные параметры
function calc(action, ...nums) {
  const initialValue = action === '*' ? 1 : 0;
  return nums.reduce((result, current) => {
    return action === '+' ? result + current : action === '*' ? result * current
: 0;
  }, initialValue)
}
console.log(calc('+', 3, 4, 21, -4)); // 24
console.log(calc('*', 1, 4, 3)); // 12
```

В описании параметров `...nums` должен всегда быть последним. Если что-то указать после него, то это вызовет ошибку:

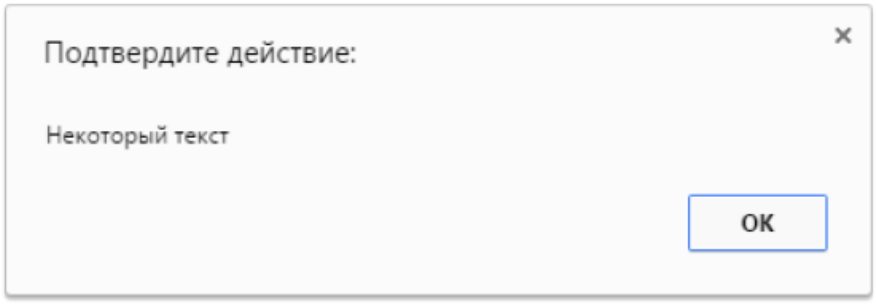
```
function calc(action, ...nums, cb) {
  // ...
}

calc(); // Uncaught SyntaxError: Rest parameter must be last formal parameter
```

Что такое встроенные (стандартные) функции

В JavaScript имеется огромный набор встроенных (стандартных) функций. Данные функции уже описаны в самом движке браузера. Практически все они являются методами того или иного объекта.

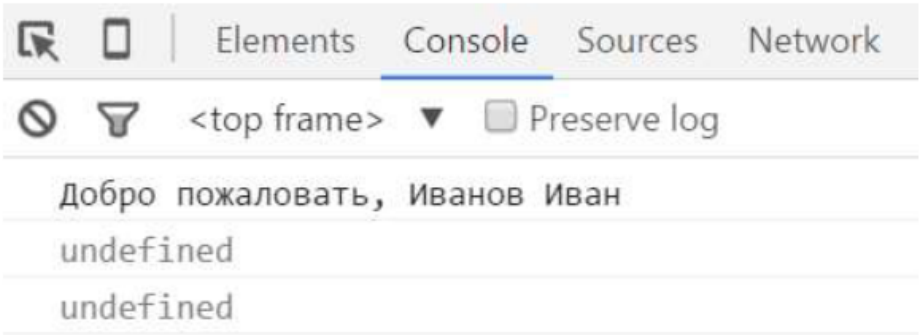
Например, для того чтобы вызвать встроенную функцию (метод) alert, её не надо предварительно объявлять. Она уже описана в браузере. Вызов метода alert осуществляется посредством указания имени, круглых скобок и аргумента внутри них. Данный метод предназначен для вывода сообщения на экран в форме диалогового окна. Текстовое сообщение берётся из значения параметра данной функции.

```
// вызов функции alert
alert("Некоторый текст");
```



Функция в JavaScript в результате своего выполнения всегда возвращает результат, даже если он явно не определён с помощью оператора `return`. Этот результат значение `undefined`.

```
// 1. функция, не возвращающая никакого результата
function sayWelcome (userFirstName, userLastName) {
  console.log("Добро пожаловать, " + userLastName + " " + userFirstName);
}
// попробуем получить результат у функции, которая ничего не возвращает
console.log(sayWelcome ("Иван", "Иванов"));
// 2. функция, содержащая оператор return без значения
function sayDay (day) {
  day = "Сегодня, " + day;
  return;
  //эта инструкция не выполнится, т.к. она идёт после оператора return
  console.log(day);
}
// попробуем получить результат у функции, которая содержит оператор return без
значения
console.log(sayDay("21 февраля 2016г."));
```



Такой же результат будет получен, если для оператора `return` не указать возвращаемое значение.