JavaScript - функции

функции, Function Declaration и Function Expression,
анонимные функции и функции callback-и,
стрелочные функции и рекурсия, чистые функции и принцип DRY

Функции в программировании

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, необходимо красиво вывести сообщение при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели – это alert(message), prompt(message, default) и confirm(question). Но можно создавать и свои.

Принцип DRY - Don't Repeat Yourself

Принцип DRY (Don't Repeat Yourself) - это принцип программирования, который призывает к избеганию дублирования кода в программном обеспечении. Суть этого принципа заключается в том, что каждый кусок информации или функциональности в программе должен иметь только одну, безусловную и авторитетную представительскую версию.

Основная идея принципа DRY заключается в том, что дублирование кода увеличивает вероятность ошибок и затрудняет его поддержку и модификацию в будущем. При нарушении принципа DRY любые изменения, вносимые в одну часть кода, могут потребовать соответствующих изменений во всех других его копиях, что может быть трудоемким и ошибочным процессом.

Применение принципа DRY в разработке программного обеспечения позволяет уменьшить сложность кода, улучшить его читаемость и облегчить его поддержку. Для соблюдения принципа DRY можно выносить повторяющийся код в отдельные функции, классы, модули или библиотеки, которые можно повторно использовать в различных частях программы. Также применение шаблонов проектирования, использование абстракций и модульного программирования помогают соблюдать принцип DRY.

Функции - базовые знания

- Объявление функции
- Локальные переменные
- Внешние переменные
- Параметры
- Значения по умолчанию
- Альтернативные параметры по умолчанию
- Возврат значения
- Выбор имени функции
- Функции == Комментарии

Объявление функции

Для создания функций мы можем использовать объявление функции.

Вначале идёт ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (в вышеприведенном примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

Наша новая функция может быть вызвана по своему имени

```
function имя(параметры) {
  ...тело...
}

function showMessage() {
  alert( 'Всем привет!' );
}

showMessage();
showMessage();
```

Локальные переменные

Переменные, объявленные внутри функции, видны только внутри этой функции.

```
function showMessage() {
  let message = "Привет, я JavaScript!"; // локальная переменная

  alert( message );
}

showMessage(); // Привет, я JavaScript!

alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции
```

Внешние переменные

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноименная переменная объявляется внутри функции, тогда она перекрывает внешнюю.

Переменные, объявленные снаружи всех функций – называются глобальными.

Глобальные переменные видимы для любой функции (если только их не перекрывают одноименные локальные переменные).

Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

```
let userName = 'Вася';

function showMessage() {
  let message = 'Привет, ' + userName;
  alert(message);
}

showMessage(); // Привет, Вася
```

```
let userName = 'Вася';

function showMessage() {
  userName = "Петя"; // (1) изменяем значение внешней переменной

  let message = 'Привет, ' + userName;
  alert(message);
}

alert( userName ); // Вася перед вызовом функции

showMessage();

alert( userName ); // Петя, значение внешней переменной было изменено функцией
```

```
let userName = "Вася";

function showMessage() {
  let userName = "Петя"; // объявляем локальную переменную

  let message = "Привет, " + userName; // Петя
  alert(message);
}

// функция создаст и будет использовать свою собственную локальную переменную userName
showMessage();
alert(userName); // Вася, не изменилась, функция не трогала внешнюю переменную
```

Параметры

Мы можем передать внутрь функции любую информацию, используя параметры.

Значение, передаваемое в качестве параметра функции, также называется аргументом.

Другими словами:

- Параметр – это переменная, указанная в круглых скобках в объявлении функции.
- Аргумент – это значение, которое передается функции при ее вызове.

Мы объявляем функции со списком параметров, затем вызываем их, передавая аргументы.

```
function showMessage(from, text) { // параметры: from, text
  alert(from + ': ' + text);
}

showMessage('Аня', 'Привет!'); // Аня: Привет! (*)
showMessage('Аня', "Как дела?"); // Аня: Как дела? (**)
```

```
function showMessage(from, text) {

  from = '*' + from + '*'; // немного украсим "from"

  alert( from + ': ' + text );
}

let from = "Аня";

showMessage(from, "Привет"); // *Аня*: Привет

// значение "from" осталось прежним, функция изменила значение локальной переменной
alert( from ); // Аня
```

Значения по умолчанию

Если при вызове функции аргумент не был указан, то его значением становится undefined.

Вычисление параметров по умолчанию
В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего аргумента.

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}

showMessage("Аня"); // Аня: текст не добавлен
```

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() выполнится только если не передан text
  // результатом будет значение text
}
```

Альтернативные параметры по умолчанию

Иногда имеет смысл присваивать значения по умолчанию для параметров не в объявлении функции, а на более позднем этапе.

Во время выполнения функции мы можем проверить, передан ли параметр, сравнив его с undefined.

Или мы можем использовать оператор ||.

Современные движки JavaScript поддерживают оператор нулевого слияния ??. Его использование будет лучшей практикой, в случае, если большинство ложных значений, таких как 0, следует расценивать как «нормальные».

```
function showMessage(text) {
  // ...
  if (text === undefined) { // если параметр отсутствует
    text = 'пустое сообщение';
  }
  alert(text);
}
showMessage(); // пустое сообщение
```

```
function showMessage(text) {
  // если значение text ложно или равняется undefined,
  text = text || 'пусто';
  ...
}
```

```
function showCount(count) {
  // если count равен undefined или null, показать "неизвестно"
  alert(count ?? "неизвестно");
}
showCount(0); // 0
showCount(null); // неизвестно
showCount(); // неизвестно
```

Возврат значения - return в функции

Функция может вернуть результат, который будет передан в вызвавший её код.

Директива return может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший ее код (присваивается переменной result выше).

Возможно использовать return и без значения. Это приведет к немедленному выходу из функции.

Результат функции с пустым return или без него – undefined
Если функция не возвращает значения, это всё равно, как если бы она возвращала undefined. Пустой return аналогичен return undefined.

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

let result = sum(1, 2);
alert( result ); // 3
```

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('А родители разрешили?');
  }
}

let age = prompt('Сколько вам лет?', 18);

if ( checkAge(age) ) {
  alert( 'Доступ получен' );
} else {
  alert( 'Доступ закрыт' );
}
```

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Вам показывается кино" ); // (*)
  // ...
}
```

Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть кратким, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление о том, что делает функция.

Функции, начинающиеся с…

- "get…" – возвращают значение,
- "calc…" – что-то вычисляют,
- "create…" – что-то создают,
- "check…" – что-то проверяют и возвращают логическое значение, и т.д.

Благодаря префиксам, при первом взгляде на имя функции становится понятным, что делает её код, и какое значение она может возвращать.

```
showMessage(..)     // показывает сообщение
getAge(..)          // возвращает возраст (получая его каким-то образом)
calcSum(..)         // вычисляет сумму и возвращает результат
createForm(..)      // создаёт форму (и обычно возвращает её)
checkPermission(..) // проверяет доступ, возвращая true/false
```

Функции == Комментарии

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Иногда следовать этому правилу непросто, но это определённо хорошее правило.

Небольшие функции не только облегчают тестирование и отладку – само существование таких функций выполняет роль хороших комментариев!

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // простое
  }
}
```

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);  // простое
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

Функция - это значение в JavaScript (особое значение)

Функция в JavaScript – это не магическая языковая структура, а особого типа значение.

В JavaScript функция – это значение, поэтому мы можем обращаться с ней как со значением.

Конечно, функция – это особое значение, в том смысле, что мы можем вызвать ее.

Но всё же это значение. Поэтому мы можем работать с ней так же, как и с другими видами значений.

Мы можем скопировать функцию в другую переменную.

Function Declaration и Function Expression

Function Declaration и Function Expression - это два способа объявления функций в JavaScript. Они имеют свои различия, особенности и применяются в разных сценариях.

Function Declaration и Function Expression предоставляют разные подходы к объявлению функций в JavaScript. Function Declaration поднимается в начало области видимости и обычно используется для определения глобальных функций, тогда как Function Expression предпочтительнее в сценариях, когда функции необходимо передавать как аргументы или сохранять в переменных. Выбор между ними зависит от конкретных требований проекта и предпочтений разработчика.

Function Declaration (Объявление функции):

Особенности:

- Может быть вызвана перед своим определением в коде (hoisted).
- Имеет доступ к переменным, объявленным в том же блоке.
- Обычно используется для определения глобальных функций или функций, объявленных на верхнем уровне файла.

Плюсы:

- Читаемость: функции удобно читать, так как их объявления находятся вверху файла.
- Доступность: функции могут быть вызваны до их определения.

Минусы:

- Ограничения: не удается использовать Function Declaration внутри блоков кода (if, for, while и т. д.).
- Не подходит для условного использования функций.

```
function functionName(parameters) {
    // тело функции
}

sayHello();

function sayHello() {
    console.log("Привет!");
}
```

Function Expression (Функциональное выражение):

Особенности:

- Не поднимается в начало области видимости, поэтому функцию можно вызвать только после ее определения.
- Может быть анонимной или именованной.
- Может быть присвоена переменной, передана как аргумент или возвращена из другой функции.

Плюсы:

- Гибкость: функции могут быть переданы в качестве аргументов другим функциям или сохранены в переменных.
- Подходит для создания функций внутри блоков кода и условий.

Минусы:

- Читаемость: объявления функций находятся в коде далее, что может ухудшить читаемость.
- Не доступны до момента определения.

```
let functionName = function(parameters) {
    // тело функции
};
```

javascript

```
let sayHello = function() {
    console.log("Привет!");
};
```

javascript

```
let greet = function sayHello() {
    console.log("Привет!");
};

greet();
```

Зачем нужна точка с запятой в конце?

У вас мог возникнуть вопрос: Почему в Function Expression ставится точка с запятой ; на конце, а в Function Declaration нет:

Ответ прост: Function Expression создается здесь как function(...) {...} внутри выражения присваивания: let sayHi = …;. Точку с запятой ; рекомендуется ставить в конце выражения, она не является частью синтаксиса функции.

Точка с запятой нужна там для более простого присваивания, такого как let sayHi = 5;, а также для присваивания функции.

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

Функции-«колбеки»

Callback-функция - это функция, которая передается в качестве аргумента другой функции и вызывается после завершения определенного действия или события.

Особенности callback-функций:

- Они позволяют выполнить асинхронные операции в JavaScript.
- Их можно использовать для обработки ответов от сервера, управления потоком выполнения и обработки событий.

```
function fetchData(callback) {
    // Какая-то логика для получения данных
    let data = "Данные";
    // Вызов callback-функции с полученными данными
    callback(data);
}

// Callback-функция, переданная в fetchData
fetchData(function(data) {
    console.log("Полученные данные:", data);
});
```

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "Вы согласны." );
}

function showCancel() {
  alert( "Вы отменили выполнение." );
}

// использование: функции showOk, showCancel передаются в качестве аргументов ask
ask("Вы согласны?", showOk, showCancel);
```

Анонимные функции:

Анонимная функция - это функция, которая не имеет имени и обычно создается на месте, где она нужна. Она может быть объявлена как функциональное выражение или использоваться как аргумент для других функций.

Особенности анонимных функций:

- Они могут быть немедленно вызваны (IIFE - Immediately Invoked Function Expression).
- Их можно передавать как аргументы другим функциям.
- Они полезны для создания функций-замыканий.

```
// Функция, принимающая callback-функцию в качестве аргумента
function processData(data, callback) {
    console.log("Обработка данных:", data);
    // Вызов callback-функции
    callback();
}

// Вызов функции processData с анонимной функцией в качестве callback-функции
processData("some data", function() {
    console.log("Callback-функция выполнена!");
});
```

```
function parentFunction() {
    console.log("Это родительская функция");

    // Анонимная функция внутри родительской функции
    let anonymousFunction = function() {
        console.log("Это анонимная функция, определенная внутри родительской функции");
    };

    // Вызов анонимной функции
    anonymousFunction();
}

// Вызов родительской функции
parentFunction();
```

стрелочные функции

Существует еще один очень простой и лаконичный синтаксис для создания функций, который часто лучше, чем Function Expression.

Он называется «функции-стрелки» или «стрелочные функции» (arrow functions)

Они бывают двух типов:

- Без фигурных скобок: (...args) => expression – правая сторона выражения: функция вычисляет его и возвращает результат. Скобки можно не ставить, если аргумент только один: n => n * 2.
- С фигурными скобками: (...args) => { body } – скобки позволяют нам писать несколько инструкций внутри функции, но при этом необходимо явно вызывать return, чтобы вернуть значение.

```
let func = (arg1, arg2, ...argN) => expression;
```

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

```
let sum = (a, b) => a + b;

/* Эта стрелочная функция представляет собой более короткую форму:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

рекурсия - теория

```
function factorial(n) {
    if (n === 0) {
        return 1; // базовый случай: факториал 0 равен 1
    } else {
        return n * factorial(n - 1); // рекурсивный случай
    }
}

console.log(factorial(5)); // Выведет: 120 (5! = 5 * 4 * 3 * 2 * 1)
```

Рекурсия - это техника программирования, при которой функция вызывает саму себя. Это позволяет решать задачи путем разбиения их на более простые версии той же задачи. Рекурсивные функции имеют базовый случай (base case), который определяет условие завершения рекурсии, и рекурсивный случай (recursive case), который вызывает функцию с более простым входом.

Плюсы рекурсии:

- Краткость и понятность кода: рекурсивные решения часто более краткие и интуитивно понятные, особенно для задач, связанных с деревьями и рекурсивными структурами данных.
- Универсальность: рекурсия может быть применена для решения широкого спектра задач.
- Меньше вероятность ошибок: правильно реализованная рекурсия может уменьшить количество ошибок, связанных с логикой циклов.

Минусы рекурсии:

- Потенциальное переполнение стека вызовов: если рекурсия уходит слишком глубоко, может произойти переполнение стека, что приведет к ошибке.
- Затраты на ресурсы: рекурсивные вызовы могут потреблять больше памяти и процессорного времени, чем итеративные циклы.
- Сложность отладки: неправильно реализованная рекурсия может быть сложной для отладки и понимания.

Рекурсивное решение:

javascript Copy code

```
function sumRecursive(n) {
    if (n === 1) {
        return 1;
    } else {
        return n + sumRecursive(n - 1);
    }
}

console.log(sumRecursive(5)); // Выведет: 15 (1 + 2 + 3 + 4 + 5)
```

Решение с использованием цикла:

javascript Copy code

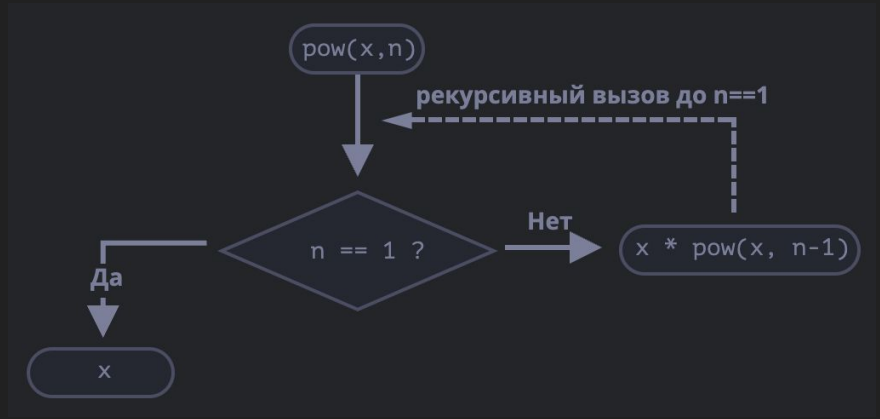
```
function sumIterative(n) {
    let sum = 0;
    for (let i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}

console.log(sumIterative(5)); // Выведет: 15 (1 + 2 + 3 + 4 + 5)
```

рекурсия - на практике

В качестве первого примера напишем функцию pow(x, n), которая возводит x в натуральную степень n. Иначе говоря, умножает x на само себя n раз.

```
function pow(x, n) {
  let result = 1;

  // умножаем result на x n раз в цикле
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

alert( pow(2, 3) ); // 8
```

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); // 8
```




```
              if n==1  = x
             /
pow(x, n) =
             \
              else     = x * pow(x, n - 1)
```

Чистые функции

Чистая функция в JavaScript - это функция, которая при заданных одинаковых входных данных всегда возвращает одинаковый результат, и не имеет побочных эффектов, то есть не изменяет состояние программы или внешние данные. При вызове чистой функции результат зависит только от переданных аргументов, и она не влияет на состояние программы за пределами своей области видимости.

Концепция:

- Однозначность (Deterministic): Чистая функция всегда возвращает одинаковый результат для одного и того же набора аргументов, без каких-либо побочных эффектов.
- Отсутствие побочных эффектов (No side effects): Чистая функция не взаимодействует с внешними переменными, файловой системой, базой данных или сетью. Она не изменяет состояние программы и не выполняет каких-либо внешних действий.

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

// Нечистая функция
let total = 0;
function addToTotal(x) {
    total += x;
    return total;
}
```

Чистые функции - плюсы и минусы

Плюсы:

- Предсказуемость и надежность: Чистые функции более предсказуемы и надежны, так как они не зависят от контекста выполнения и внешних данных.
- Тестируемость: Поскольку результат работы чистой функции зависит только от ее входных данных, ее легко тестировать, что помогает обнаружить и исправить ошибки.
- Параллелизация и оптимизация: Чистые функции могут быть безопасно вызваны параллельно, так как они не изменяют общее состояние программы. Они также могут быть легче оптимизированы и кэшированы.

Минусы:

- Ограничения: Некоторые задачи, такие как взаимодействие с пользовательским интерфейсом или изменение глобального состояния, требуют использования нечистых функций.
- Сложность в реализации: Некоторые задачи могут быть сложны для реализации с использованием только чистых функций, что может потребовать больше усилий и времени.
- Производительность: Иногда использование только чистых функций может привести к увеличению количества копий данных, что может негативно сказаться на производительности.

Несмотря на свои ограничения, использование чистых функций в JavaScript может повысить качество кода, сделать его более надежным и легким для понимания, а также обеспечить более прозрачное и предсказуемое поведение программы.

Ресурсы:

Функции - ТЫК

Function Expression - ТЫК

Стрелочные функции - ТЫК

Рекурсия - ТЫК