Функции

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.

Объявление функции

Для создания функций мы можем использовать *объявление функции*.

Пример объявления функции:

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

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

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

Наша новая функция может быть вызвана по её имени: `showMessage()`.

Например:

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

showMessage();
showMessage();
```

Вызов `showMessage()` выполняет код функции. Здесь мы увидим сообщение дважды.

Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода.

Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

Локальные переменные

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

Например:

```
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

У функции есть доступ к внешним переменным, например:

```
let userName = 'Вася';

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

showMessage(); // Привет, Вася
```

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

Например:

```
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
```

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

Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную `userName`. Внешняя будет проигнорирована:

```
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
```

Глобальные переменные

Переменные, объявленные снаружи всех функций, такие как внешняя переменная `userName` в вышеприведённом коде – называются *глобальными*.

Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые 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.

Параметры

Мы можем передать внутрь функции любую информацию, используя параметры (также называемые *аргументами функции*).

В нижеприведённом примере функции передаются два параметра: `from` и `text`.

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

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

Когда функция вызывается в строках `(*)` и `(**)`, переданные значения копируются в локальные переменные `from` и `text`. Затем они используются в теле функции.

Вот ещё один пример: у нас есть переменная `from`, и мы передаём её функции. Обратите внимание: функция изменяет значение `from`, но это изменение не видно снаружи. Функция всегда получает только копию значения:

```
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
```

Параметры по умолчанию

Если параметр не указан, то его значением становится `undefined`.

Например, вышеупомянутая функция `showMessage(from, text)` может быть вызвана с одним аргументом:

```
showMessage("Аня");
```

Это не приведёт к ошибке. Такой вызов выведет `"Аня: undefined"`. В вызове не указан параметр `text`, поэтому предполагается, что `text === undefined`.

Если мы хотим задать параметру `text` значение по умолчанию, мы должны указать его после `=`:

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

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

Теперь, если параметр `text` не указан, его значением будет `"текст не добавлен"`

В данном случае `"текст не добавлен"` это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра. Например:

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

Вычисление параметров по умолчанию

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра.

В примере выше `anotherFunction()` будет вызываться каждый раз, когда `showMessage()` вызывается без параметра `text`.

Использование параметров по умолчанию в ранних версиях JavaScript

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах.

Например, явная проверка на `undefined`:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'текст не добавлен';
  }

  alert( from + ": " + text );
}
```

...Или с помощью оператора `||`:

```
function showMessage(from, text) {
  // Если значение text ложно, тогда присвоить параметру text значение по умолчанию
  text = text || 'текст не добавлен';
  ...
}
```

Возврат значения

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

Простейшим примером может служить функция сложения двух чисел:

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

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

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

Вызовов `return` может быть несколько, например:

```
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' );
}
```

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

Например:

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

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

В коде выше, если `checkAge(age)` вернёт `false`, `showMovie` не выполнит `alert`.

Результат функции с пустым `return` или без него – `undefined`

Если функция не возвращает значения, это всё равно, как если бы она возвращала `undefined`:

```
function doNothing() { /* пусто */ }

alert( doNothing() === undefined ); // true
```

Пустой `return` аналогичен `return undefined`:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

Никогда не добавляйте перевод строки между `return` и его значением

Для длинного выражения в `return` может быть заманчиво разместить его на нескольких отдельных строках, например так:

```
return
 (some + long + expression + or + whatever * f(a) + f(b))
```

Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после `return`. Для него это будет выглядеть так:

```
return;
 (some + long + expression + or + whatever * f(a) + f(b))
```

Таким образом, это фактически стало пустым `return`.

Если мы хотим, чтобы возвращаемое выражение занимало несколько строк, нужно начать его на той же строке, что и `return`. Или, хотя бы, поставить там открывающую скобку, вот так:

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

И тогда всё сработает, как задумано.

Выбор имени функции

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

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов.

Например, функции, начинающиеся с `"show"` обычно что-то показывают.

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.

Примеры таких имён:

```
showMessage(..)     // показывает сообщение
getAge(..)          // возвращает возраст (в каком либо значении)
calcSum(..)         // вычисляет сумму и возвращает результат
createForm(..)      // создаёт форму (и обычно возвращает её)
checkPermission(..) // проверяет доступ, возвращая true/false
```

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

Одна функция – одно действие

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием.

Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе (в этом случае мы можем создать третью функцию, которая будет их вызывать).

Несколько примеров, которые нарушают это правило:

- `getAge` – будет плохим выбором, если функция будет выводить `alert` с возрастом (должна только возвращать его).
- `createForm` – будет плохим выбором, если функция будет изменять документ, добавляя форму в него (должна только создавать форму и возвращать её).
- `checkPermission` – будет плохим выбором, если функция будет отображать сообщение с текстом `доступ разрешён/запрещён` (должна только выполнять проверку и возвращать её результат).

В этих примерах использовались общепринятые смыслы префиксов. Конечно, вы в команде можете договориться о других значениях, но обычно они мало отличаются от общепринятых. В любом случае вы и ваша команда должны точно понимать, что значит префикс, что функция с ним может делать, а чего не может.

Сверхкороткие имена функций

Имена функций, которые используются *очень часто*, иногда делают сверхкороткими.

Например, во фреймворке jQuery есть функция с именем `$`. В библиотеке Lodash основная функция представлена именем `_`.

Это исключения. В основном имена функций должны быть в меру краткими и описательными.

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!

Например, сравним ниже две функции `showPrimes(n)`. Каждая из них выводит простое число до `n`.

Первый вариант использует метку `nextPrime`:

```
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
  }
}
```

Второй вариант использует дополнительную функцию `isPrime(n)` для проверки на 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;
}
```

Второй вариант легче для понимания, не правда ли? Вместо куска кода мы видим название действия (`isPrime`). Иногда разработчики называют такой код *самодокументируемым*.

Таким образом, допустимо создавать функции, даже если мы не планируем повторно использовать их. Такие функции структурируют код и делают его более понятным.

Итого

Объявление функции имеет вид:

```
function имя(параметры, через, запятую) {
  /* тело, код функции */
}
```

- Передаваемые значения копируются в параметры функции и становятся локальными переменными.
- Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.
- Функция может возвращать значение. Если этого не происходит, тогда результат равен `undefined`.

Для того, чтобы сделать код более чистым и понятным, рекомендуется использовать локальные переменные и параметры функций, не пользоваться внешними переменными.

Функция, которая получает параметры, работает с ними и затем возвращает результат, гораздо понятнее функции, вызываемой без параметров, но изменяющей внешние переменные, что чревато побочными эффектами.

Именование функций:

- Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает, и что возвращает.
- Функция – это действие, поэтому её имя обычно является глаголом.
- Есть много общепринятых префиксов, таких как: `create…`, `show…`, `get…`, `check…` и т.д. Пользуйтесь ими как подсказками, поясняющими, что делает функция.

Функции являются основными строительными блоками скриптов. Мы рассмотрели лишь основы функций в JavaScript, но уже сейчас можем создавать и использовать их. Это только начало пути. Мы будем неоднократно возвращаться к функциям и изучать их всё более и более глубоко.