**Function Expression**

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

Синтаксис, который мы использовали до этого, называется *Function Declaration* (Объявление Функции):

function sayHi() {

alert( "Привет" );

}

Существует ещё один синтаксис создания функций, который называется *Function Expression* (Функциональное Выражение).

Оно выглядит вот так:

let sayHi = function() {

alert( "Привет" );

};

В коде выше функция создаётся и явно присваивается переменной, как любое другое значение. По сути без разницы, как мы определили функцию, это просто значение, хранимое в переменной sayHi.

Смысл обоих примеров кода одинаков: "создать функцию и поместить её значение в переменную sayHi".

Мы можем даже вывести это значение с помощью alert:

function sayHi() {

alert( "Привет" );

}

alert( sayHi ); // выведет код функции

Обратите внимание, что последняя строка не вызывает функцию sayHi, после её имени нет круглых скобок. Существуют языки программирования, в которых любое упоминание имени функции совершает её вызов. JavaScript – не один из них.

В JavaScript функции – это значения, поэтому мы и обращаемся с ними, как со значениями. Код выше выведет строковое представление функции, которое является её исходным кодом.

Конечно, функция – не обычное значение, в том смысле, что мы можем вызвать его при помощи скобок: sayHi().

Но всё же это значение. Поэтому мы можем делать с ним то же самое, что и с любым другим значением.

Мы можем скопировать функцию в другую переменную:

function sayHi() { // (1) создаём

alert( "Привет" );

}

let func = sayHi; // (2) копируем

func(); // Привет // (3) вызываем копию (работает)!

sayHi(); // Привет // прежняя тоже работает (почему бы нет)

Давайте подробно разберём всё, что тут произошло:

1. Объявление Function Declaration (1) создало функцию и присвоило её значение переменной с именем sayHi.
2. В строке (2) мы скопировали её значение в переменную func. Обратите внимание (ещё раз): нет круглых скобок после sayHi. Если бы они были, то выражение func = sayHi() записало бы *результат вызова* sayHi() в переменную func, а не саму *функцию* sayHi.
3. Теперь функция может быть вызвана с помощью обеих переменных sayHi() и func().

Заметим, что мы могли бы использовать и Function Expression для того, чтобы создать sayHi в первой строке:

let sayHi = function() {

alert( "Привет" );

};

let func = sayHi;

// ...

Результат был бы таким же.

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:

function sayHi() {

// ...

}

let sayHi = function() {

// ...

};

Ответ прост:

* Нет необходимости в ; в конце блоков кода и синтаксических конструкций, которые их используют, таких как if { ... }, for { }, function f { } и т.д.
* Function Expression использует внутри себя инструкции присваивания let sayHi = ...; как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой не относится непосредственно к Function Expression, она лишь завершает инструкцию.

## Функции-«колбэки»

Рассмотрим ещё примеры функциональных выражений и передачи функции как значения.

Давайте напишем функцию ask(question, yes, no) с тремя параметрами:

question

Текст вопроса

yes

Функция, которая будет вызываться, если ответ будет «Yes»

no

Функция, которая будет вызываться, если ответ будет «No»

Наша функция должна задать вопрос question и, в зависимости от того, как ответит пользователь, вызвать yes() или no():

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);

На практике подобные функции очень полезны. Основное отличие «реальной» функции ask от примера выше будет в том, что она использует более сложные способы взаимодействия с пользователем, чем простой вызов confirm. В браузерах такие функции обычно отображают красивые диалоговые окна. Но это уже другая история.

**Аргументы функции ask ещё называют *функциями-колбэками* или просто *колбэками*.**

Ключевая идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо. В нашем случае, showOk становится *колбэком*’ для ответа «yes», а showCancel – для ответа «no».

Мы можем переписать этот пример значительно короче, используя Function Expression:

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

ask(

"Вы согласны?",

function() { alert("Вы согласились."); },

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

);

Здесь функции объявляются прямо внутри вызова ask(...). У них нет имён, поэтому они называются *анонимными*. Такие функции недоступны снаружи ask (потому что они не присвоены переменным), но это как раз то, что нам нужно.

Подобный код, появившийся в нашем скрипте выглядит очень естественно, в духе JavaScript.

Функция – это значение, представляющее «действие»

Обычные значения, такие как строки или числа представляют собой *данные*.

Функции, с другой стороны, можно воспринимать как «действия».

Мы можем передавать их из переменной в переменную и запускать, когда захотим.

## Function Expression в сравнении с Function Declaration

Давайте разберём ключевые отличия Function Declaration от Function Expression.

Во-первых, синтаксис: как определить, что есть что в коде.

* Function Declaration: функция объявляется отдельной конструкцией «function…» в основном потоке кода.
* // Function Declaration
* function sum(a, b) {
* return a + b;

}

Function Expression: функция, созданная внутри другого выражения или синтаксической конструкции. В данном случае функция создаётся в правой части «выражения присваивания» =:

// Function Expression

let sum = function(a, b) {

return a + b;

};

Более тонкое отличие состоит, в том, *когда* создаётся функция движком JavaScript.

**Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться.**

После того, как поток выполнения достигнет правой части выражения присваивания let sum = function… – с этого момента, функция считается созданной и может быть использована (присвоена переменной, вызвана и т.д. ).

С Function Declaration всё иначе.

**Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке).**

Другими словами, когда движок JavaScript *готовится* выполнять скрипт или блок кода, прежде всего он ищет в нём Function Declaration и создаёт все такие функции. Можно считать этот процесс «стадией инициализации».

И только после того, как все объявления Function Declaration будут обработаны, продолжится выполнение.

В результате, функции, созданные, как Function Declaration могут быть вызваны раньше своих определений.

Например, так будет работать:

sayHi("Вася"); // Привет, Вася

function sayHi(name) {

alert( `Привет, ${name}` );

}

Функция sayHi была создана, когда движок JavaScript подготавливал скрипт к выполнению, и такая функция видна повсюду в этом скрипте.

…Если бы это было Function Expression, то такой код вызовет ошибку:

sayHi("Вася"); // ошибка!

let sayHi = function(name) { // (\*) магии больше нет

alert( `Привет, ${name}` );

};

Функции, объявленные при помощи Function Expression, создаются тогда, когда выполнение доходит до них. Это случится только на строке, помеченной звёздочкой (\*). Слишком поздно.

Ещё одна важная особенность Function Declaration заключается в их блочной области видимости.

**В строгом режиме, когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока. Но не снаружи него.**

Для примера давайте представим, что нам нужно создать функцию welcome() в зависимости от значения переменной age, которое мы получим во время выполнения кода. И затем запланируем использовать её когда-нибудь в будущем.

Такой код, использующий Function Declaration, работать не будет:

let age = prompt("Сколько Вам лет?", 18);

// в зависимости от условия объявляем функцию

if (age < 18) {

function welcome() {

alert("Привет!");

}

} else {

function welcome() {

alert("Здравствуйте!");

}

}

// ...не работает

welcome(); // Error: welcome is not defined

Это произошло, так как объявление Function Declaration видимо только внутри блока кода, в котором располагается.

Вот ещё один пример:

let age = 16; // присвоим для примера 16

if (age < 18) {

welcome(); // \ (выполнится)

// |

function welcome() { // |

alert("Привет!"); // | Function Declaration доступно

} // | во всём блоке кода, в котором объявлено

// |

welcome(); // / (выполнится)

} else {

function welcome() {

alert("Здравствуйте!");

}

}

// здесь фигурная скобка закрывается,

// поэтому Function Declaration, созданные внутри блока кода выше -- недоступны отсюда.

welcome(); // Ошибка: welcome is not defined

Что можно сделать, чтобы welcome была видима снаружи if?

Верным подходом будет воспользоваться функцией, объявленной при помощи Function Expression, и присвоить значение welcome переменной, объявленной снаружи if, что обеспечит нам нужную видимость.

Такой код работает, как ожидалось:

let age = prompt("Сколько Вам лет?", 18);

let welcome;

if (age < 18) {

welcome = function() {

alert("Привет!");

};

} else {

welcome = function() {

alert("Здравствуйте!");

};

}

welcome(); // теперь всё в порядке

Можно упростить этот код ещё сильнее, используя условный оператор ?:

let age = prompt("Сколько Вам лет?", 18);

let welcome = (age < 18) ?

function() { alert("Привет!"); } :

function() { alert("Здравствуйте!"); };

welcome(); // теперь всё в порядке

Когда использовать Function Declaration, а когда Function Expression?

Как правило, если нам понадобилась функция, в первую очередь нужно рассматривать синтаксис Function Declaration, который мы использовали до этого. Он даёт нам больше свободы в том, как мы можем организовывать код. Функции, объявленные таким образом, можно вызывать до их объявления.

Также функции вида function f(…) {…} чуть более заметны в коде, чем let f = function(…) {…}. Function Declaration легче «ловятся глазами».

…Но если Function Declaration нам не подходит по какой-то причине (мы рассмотрели это в примере выше), то можно использовать объявление при помощи Function Expression.

## Итого

* Функции – это значения. Они могут быть присвоены, скопированы или объявлены в другом месте кода.
* Если функция объявлена как отдельная инструкция в основном потоке кода, то это Function Declaration.
* Если функция была создана как часть выражения, то считается, что эта функция объявлена при помощи Function Expression.
* Function Declaration обрабатываются перед выполнением блока кода. Они видны во всём блоке.
* Функции, объявленные при помощи Function Expression, создаются, только когда поток выполнения достигает их.

В большинстве случаев, когда нам нужно создать функцию, предпочтительно использовать Function Declaration, т.к. функция будет видима до своего объявления в коде. Это позволяет более гибко организовывать код, и улучшает его читаемость.

Таким образом, мы должны прибегать к объявлению функций при помощи Function Expression в случае, когда синтаксис Function Declaration не подходит для нашей задачи. Мы рассмотрели несколько таких примеров в этой главе, и рассмотрим их ещё больше в будущем.

**Функции-стрелки, основы**

Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression.

Он называется «функции-стрелки» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

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

…Такой код создаёт функцию func с аргументами arg1..argN и вычисляет expression с правой стороны с их использованием, возвращая результат.

Другими словами, это более короткий вариант такой записи:

let func = function(arg1, arg2, ...argN) {

return expression;

};

Давайте взглянем на конкретный пример:

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

/\* Более короткая форма для:

let sum = function(a, b) {

return a + b;

};

\*/

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

То есть, (a, b) => a + b задаёт функцию с двумя аргументами a и b, которая при запуске вычисляет выражение справа a + b и возвращает его результат.

* Если у нас только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:
* // тоже что и
* // let double = function(n) { return n \* 2 }
* let double = n => n \* 2;

alert( double(3) ); // 6

Если нет аргументов, указываются пустые круглые скобки:

let sayHi = () => alert("Hello!");

sayHi();

Функции-стрелки могут быть использованы так же, как и Function Expression.

Например, для динамического создания функции:

let age = prompt("Сколько Вам лет?", 18);

let welcome = (age < 18) ?

() => alert('Привет') :

() => alert("Здравствуйте!");

welcome(); // теперь всё в порядке

Поначалу функции-стрелки могут показаться необычными и трудночитаемыми, но это быстро пройдёт, как только глаза привыкнут к этим конструкциям.

Они очень удобны для простых однострочных действий, когда лень писать много букв.

## Многострочные стрелочные функции

В примерах выше аргументы использовались слева от =>, а справа вычислялось выражение с их значениями.

Порой нам нужно что-то посложнее, например, выполнить несколько инструкций. Это также возможно, нужно лишь заключить инструкции в фигурные скобки. И использовать return внутри них, как в обычной функции.

Например:

let sum = (a, b) => { // фигурная скобка, открывающая тело многострочной функции

let result = a + b;

return result; // при фигурных скобках для возврата значения нужно явно вызвать return

};

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

Дальше будет ещё информация

Здесь мы рассмотрели функции-стрелки как способ писать меньше букв. Но это далеко не всё!

Стрелочные функции обладают другими интересными особенностями. Их изучение требует знания некоторых других возможностей языка JavaScript, поэтому мы вернёмся к стрелочным функциям позже, в главе [Повторяем стрелочные функции](https://learn.javascript.ru/arrow-functions).

А пока мы можем использовать их для простых однострочных действий и колбэков.

## Итого

Функции-стрелки очень удобны для однострочных действий. Они бывают двух типов:

1. Без фигурных скобок: (...args) => expression – правая сторона выражение: функция выполняет его и возвращает результат.
2. С фигурными скобками: (...args) => { body } – скобки позволяют нам писать многострочные инструкции внутри функции, но при этом необходимо указывать директиву return, чтобы вернуть какое-либо значение.