Введение в функции

Кирилл Талецкий

TeachMeSkills
3 августа 2023

Что будет

- Объявление и вызов
- Параметры и аргументы
- Ключевое слово return
- Функциональные выражения
- **Перерыв**
- Функция как значение
- Введение в стрелочные функции

Функции

- Позволяют выделять повторяющийся функционал
- Мы уже использовали встроенные в JS функции
  - alert(), console.log(), Number()

Объявление

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

```
function log(message) {
  console.log(message)
}

log()
log('hey there');
log('how is it going?')
```

Переменные

- Внутри функции можно объявить любые переменные - они будут видны только внутри этой функции

```
function log(message) {
  const prefix = 'Logger says';

  console.log(`${prefix}: ${message}`);
}

log('successfully connected');

console.log(prefix) // reference error - переменная prefix видна только внутри функции
```

Переменные

Функции имеют доступ к внешним переменным

```
let count = 0;

function logCount() {
  console.log(count);
}

logCount(); // 0
count++;
logCount(); // 1
```

Функции имеют возможность менять внешние переменные

```
function increment() {
  count++
}

logCount(); // 1
increment();
logCount(); // 2
```

Функция сначала ищет переменные внутри себя, затем — снаружи

```
const hello = 'Hello from outer scope';

function greet() {
  const hello = 'Hello from function';

  console.log(hello)
}

greet() // Hello from function
```

Параметры

Параметры

- В функцию можно передать любые параметры

```
function log(from, message) {
  console.log(`${from}: ${message}`);
}

log('Auth Service', 'permission denied'); // Auth Service: permission denied
```

Параметры

- Значения будут скопированы, то есть функция не имеет доступ к переданным переменным, только к значениям

```
function log(from, message) {
  from = `Message from ${from}`;
  console.log(`${from}: ${message}`);
}

const SERVICE_NAME = 'Auth Service';

log(SERVICE_NAME, 'permission denied') // Message from Auth Service: permission denied
console.log(SERVICE_NAME) // Auth Service
```

Параметры

- Если аргумент не был передан, то он вычислится как `undefined`

```
function log(from, message) {
  console.log(`${from}: ${message}`);
}

log('Some Service'); // Some Service: undefined
```

Параметры

- Чтобы обработать отсутствие аргумента, можно задать значение по умолчанию

```
function log(from, message = 'empty message') {
  console.log(`${from}: ${message}`);
}

log('Some Service'); // Some Service: empty message
```

- Значением по умолчанию может быть вычислено с помощью любого JS выражения
- Обработано будет только значение `undefined` (отсутствие аргумента)

Параметры

- Значение будет присвоено, только если соответствующий аргумент не был передан. Если вы ожидаете, что будет передан `null` или другое неустраивающее вас значение, то это лучше обработать отдельно.

```
function log(from, message) {
  from = from ?? 'Unknown said';
  message = message ?? 'empty message';

  console.log(`${from}: ${message}`);
}
```

Возвращаемое значение

Возвращаемое значение

- Ключевое слово `return` позволяет вернуть из функции значение

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

console.log(sum(2, 2)); // 4
```

Возвращаемое значение

- `return` немедленно прекращает выполнение функции и возвращает значение

```
function sum(a, b) {
  if (isNaN(a) && isNaN(b)) return 0;
  if (isNaN(a)) return b;
  if (isNaN(b)) return a;
  return a + b;
}

console.log(sum(7, 'kek')) // 7
console.log(sum('lol', 42)) // 42
console.log(sum(2, 2)) // 4
```

Возвращаемое значение

- Возможна запись `return` без возвращаемого значения — тогда выполнение будет прервано, а функция вернёт `undefined`

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

console.log(sum(7, 'kek')) // undefined
console.log(sum('lol', 42)) // undefined
console.log(sum(2, 2)) // 4
```

Возвращаемое значение

- Если в функции не будет выполнен `return`, то она так же вернёт `undefined`

```
function showMessage() {
  alert('message');
}

console.log(showMessage()); // undefined
```

Function Expression

Function Expression

- Ещё один синтаксис создания функций

```
const handleError = function (message) {
  console.error(`An error occured: ${message}`)
}

handleError('wrong input'); // An error occured: wrong input
```

Function Expression

```
const handleError = function (message) {
  console.error(`An error occured: ${message}`)
}
```

```
const handleError =   // (2) присваивание функции в переменную `handleError`
                      // ⬆⬆
function (message) {  // (1) объявление анонимной функции
  console.error(`An error occured: ${message}`)
}
```

Function Declaration vs Function Expression

Function Expression

Функциональное выражение - справа от оператора присваивания

```
const handleError = function (message) {
  console.error(`An error occured: ${message}`)
}
```

Function Declaration

Объявление функции - специальная конструкция языка

```
function handleError(message) {
  console.error(`An error occured: ${message}`)
}
```

Function Declaration vs Function Expression

- Function Expression — функция создаётся только в момент, когда исполняется соответствующий код
- Так же как и обычные let/const переменные, она “всплывёт”, но к ней нельзя будет обратиться (поэтому обычно говорят, что FE не всплывает, хотя технически это не так)

```
sum(2, 2) // выведет ошибку - переменная sum существует, но к ней нельзя обратиться, т.к. она объявлена через const (let)

const sum = function (a, b) { // только в этот момент функция будет создана и присвоена в переменную
  return a + b;
}

sum(2, 2) // 4 - переменная sum существует, к ней можно обратиться, в ней есть функция, она будет вызвана
```

Function Declaration vs Function Expression

- Function Declaration — функция создаётся в момент, когда интерпретатор первый раз читает блок кода, то есть до исполнения блока.
- К функции можно обратиться до её объявления — она “всплывает”

```
/**
 * Сначала JS пройдётся по блоку, найдёт все var, function
 */

sum(2, 2) // 4 - переменная sum существует, к ней можно обратиться, в ней есть функция, она будет вызвана

function sum(a, b) { // только в этот момент функция будет создана и присвоена в переменную
  return a + b;
}

sum(2, 2) // 4 - переменная sum существует, к ней можно обратиться, в ней есть функция, она будет вызвана
```

Function Declaration vs Function Expression

- Function Declaration не всплывает из блоков кода, других функций

```
sum(2, 2) // ошибка – JS не знает, что такое `sum`

{
  function sum(a, b) { // только в этот момент функция будет создана и присвоена в переменную
    return a + b;
  }
}

sum(2, 2) // 4 – переменная sum существует, к ней можно обратиться, в ней есть функция, она будет вызвана
```

Function Declaration в ветвлении

```
const condition = false;

logCondition();

if (condition) {
  function logCondition() {
    console.log('true!')
  }
} else {
  function logCondition() {
    console.log('false!')
  }
}
```

Function Declaration в ветвлении

```
const condition = Math.random(1) > 0.5;

logCondition();

if (condition) {    // (1) JS не знает заранее, будет ли тут true или false
                    // чтобы это узнать, пришлось бы выполнить код выше, а до этого ещё не дошёл ход
  function logCondition() {
    console.log('true!')
  }
} else {
  function logCondition() {
    console.log('false!')
  }
}
```

- Function Declaration не всплывает из ветвлений (не из-за блочной области видимости)

Перерыв

Функция как значение

Функции как значения

- В JavaScript функция — это *значение*
  - Такое же, как и `42`, `5 > 1`, `x === ‘hi’` и т.д.
  - Любую функцию можно присвоить в переменную
  - Любое function declaration можно переприсвоить
  - К функциям можно применить любой оператор

Функции как значения

- Так как функции, это значения они могут быть частью любых JS выражений
  - Функции можно переприсваивать в переменные

```
let showMessage = function (message) {
  console.log(message, message, message, message, message);
}

showMessage('hi there!'); // выведет 5 записей в консоль

showMessage = alert;
showMessage('hi there'); // покажет модалку alert

showMessage = console.log
showMessage('hi there'); // выведет сообщение в консоль
```

Функции как значения

- Так как функции, это значения они могут быть частью любых JS выражений
  - Функции можно использовать в логических выражениях

```
const handleError = function() {
  console.error(`An error occured`);
}

const handleSuccess = function() {
  console.log(`Succesfully connected`);
}

const isError = true;
const handler = isError ? handleError : handleSuccess;

handler();
```

Функции как значения

- Функции можно передавать как аргумент в другие функции — callback (функции обратного вызова)

```
function makeRequest(url, onSuccess) {
  // make request to server via url
  // ...

  onSuccess();
}

const handleSuccess = function () {
  console.log('успешно запросили!')
}

makeRequest('https://google.com', handleSuccess); // успешно запросили!
```

Колбэки

- Более короткая запись

```
function makeRequest(url, onSuccess) {
  // make request to server via url
  // ...

  onSuccess();
}

makeRequest('https://google.com', function () {
  console.log('успешно запросили!')
})
```

Функции как значения

- Функции можно возвращать из других функций — higher-order function (функции высшего порядка)

```
function fetchData(url) {
  // притворимся, что функция запрашивает данные

  // возвращаем фейковый ответ с сервера
  return 'status - 200, success'
}

fetchData('https://google.com') // мы сделали запрос, ничего не вывели в консоль

function withLogger(func) {
  return function (argument) {
    console.log(`Function was called with argument: ${argument}`);
    return func(argument);
  }
}

const loggedFetchData = withLogger(fetchData);

loggedFetchData('https://google.com') // https://google.com
// мы сделали запрос, вывели в консоль аргумерт
```

Стрелочные функции

Введение в стрелочные функции

- Более лаконичный синтаксис function expression:

```
const func = (аргументы, через, запятую) => возвращаемое_значение;
```

```
const greet = () => console.log('Hi there!');
greet();

const sum = (a, b) => a + b;
const mul = (a, b, c, d) => a * b * c * d;
const log = (message) => console.log(message);
```

Введение в стрелочные функции

- Пример с колбэками

```
function makeRequest(url, onSuccess) {
  // make request to server via url
  // ...

  onSuccess();
}

makeRequest('https://google.com', function () {
  console.log('успешно запросили!')
})
```

```
function makeRequest(url, onSuccess) {
  // make request to server via url
  // ...

  onSuccess();
}

makeRequest('https://google.com', () => console.log('успешно запросили!'))
```

Введение в стрелочные функции

- Стрелочные функции можно сделать многострочными

```
const multiline = () => {
  console.log('Это действие на строке 1');
  console.log('Это действие на строке 2');
  console.log('Это действие на строке 3');

  return 5 + 2; // можно вернуть значение
}
```

- **У стрелочных функций есть дополнительные особенности, но о них мы поговорим после того как познакомимся с объектами**

Дополнение

GitHub PR Flow

Чистые функции

- Функция считается чистой, если она
  - При одинаковых параметрах выдаёт одинаковый результат — детерминирована
  - Не имеет побочных эффектов

- В продуктовом JS коде принято по умолчанию использовать чистые функции — их легко читать, понимать и тестировать
- Совет: старайтесь вообще не писать функции, которые изменяют внешние переменные

Dependency Injection

Callback hell