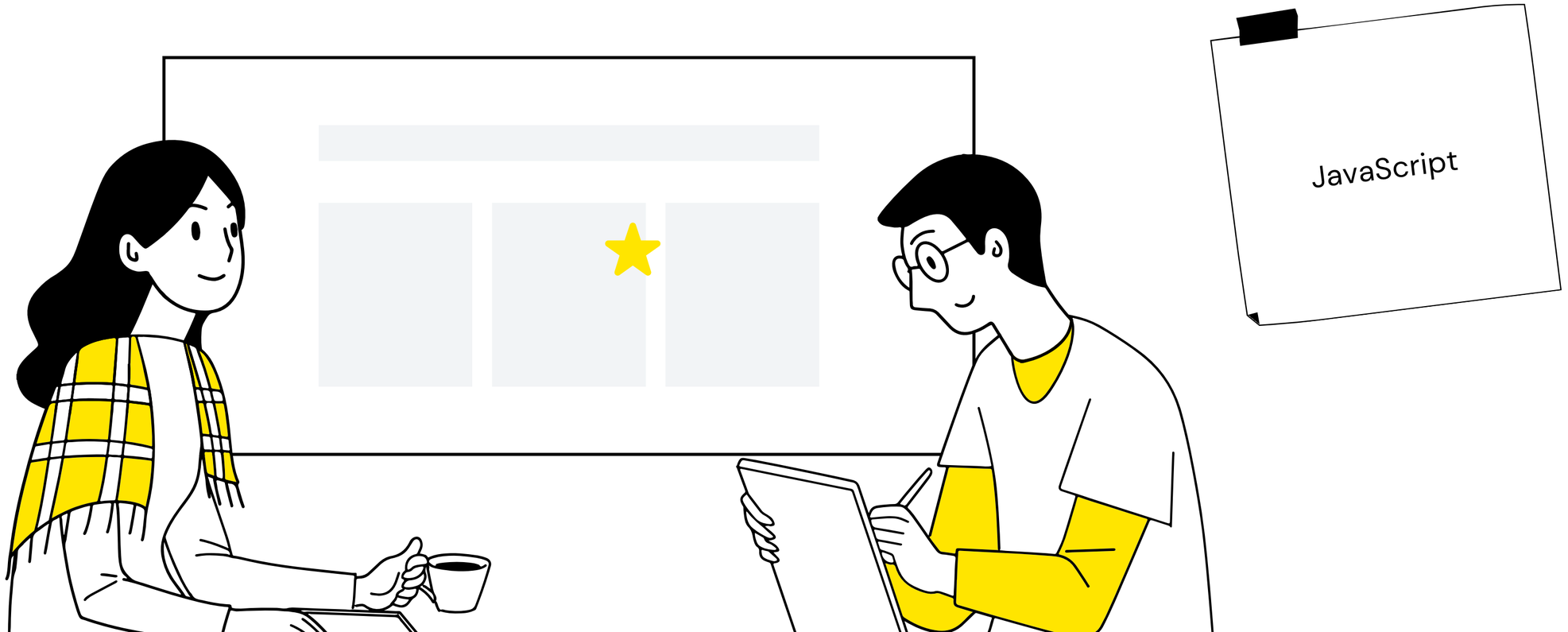
JavaScript



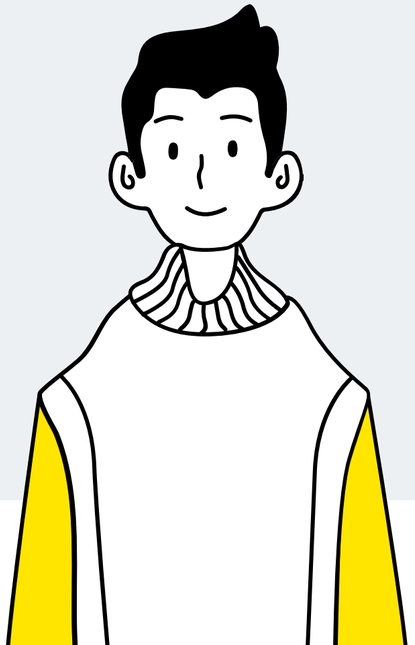
План

1. Функции, область видимости
2. Function expression, declaration
3. Callback
4. Стрелочные функции

Функции

Функции в JavaScript представляют собой блоки кода, которые могут быть вызваны для выполнения определенной задачи. Они используются для организации кода, повторного использования и абстрагирования логики программы.

```
// Объявление функции
function sayHello() {
  console.log('Привет, мир!');
}
sayHello(); // Выведет в консоль: "Привет, мир!"
```



Для чего нужны функции?

Функции играют ключевую роль в программировании, и вот некоторые причины, по которым они важны:

1. Многократное использование кода: Если у вас есть задача, которую нужно выполнить многократно в разных частях вашего кода, вы можете поместить эту задачу в функцию и вызывать эту функцию каждый раз, когда нужно выполнить задачу. Это сокращает дублирование кода.
2. Модульность: Функции позволяют разбить большие, сложные задачи на более мелкие и управляемые части. Каждая функция занимается выполнением своей собственной задачи.
3. Абстракция: Когда вы используете функцию, вам не всегда нужно знать, что происходит внутри. Вы знаете, что входит в функцию и что из нее выходит, и это часто всё, что вам нужно знать.

Параметры и аргументы функции

Функции могут принимать "параметры". Параметры – это переменные, которые используются внутри функции и получают свое значение при вызове функции. Значения, которые вы передаете в функцию при вызове, называются "аргументами".

```
// параметры функции
function greet(name) { // "name" - это параметр функции
  console.log('Привет, ' + name + '!');
}

greet( name: 'Анна'); // "Анна" - это аргумент функции
```

Этот код объявляет функцию greet, которая принимает один параметр (name), и затем вызывает эту функцию, передавая ей аргумент ("Анна"). В результате в консоли будет напечатано: "Привет, Анна!".

Параметры по умолчанию

Параметры по умолчанию в JavaScript позволяют устанавливать начальные значения для аргументов функции, которые могут быть переопределены при вызове функции. Если функция вызывается без указания одного или нескольких аргументов, то вместо них будут использоваться значения по умолчанию.

```
// параметры по умолчанию
function greet(name : string  = 'друг') {
  console.log(`Привет, ${name}!`);
}

greet( name: 'Анна'); // Привет, Анна!
greet(); // Привет, друг!
```

В этом примере, если вызвать функцию greet() без аргумента, параметр name будет использовать значение 'друг' по умолчанию.

Возврат значения

Функции в JavaScript могут возвращать значение, которое затем можно использовать в другой части программы. Это делается с помощью оператора return.
Когда функция достигает оператора return, она останавливается и возвращает значение, указанное после return.

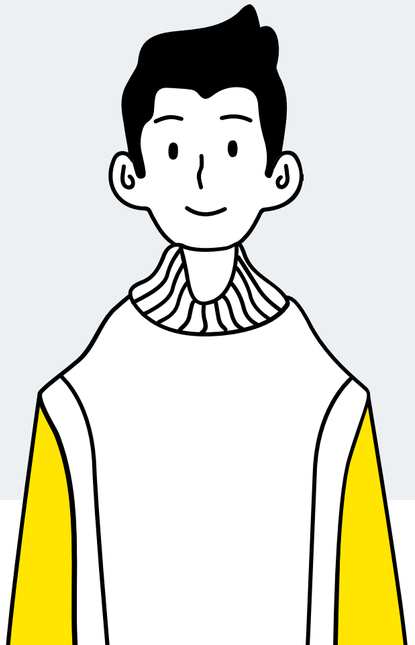
```
//возврат значения
function addNumbers(a, b) {
  return a + b;
}

let sum = addNumbers( a: 3,  b: 4);
console.log(sum); // Выведет в консоль: 7
```

В этом примере функция addNumbers принимает два параметра (a и b), складывает их и возвращает результат. Значение, возвращаемое функцией, затем сохраняется в переменной sum.

Важно помнить, что оператор return также означает выход из функции. Это означает, что как только выполнение функции достигает оператора return, она прекращается и контролирует возвращение вызывающему коду. Любой код после оператора return в функции будет игнорироваться.

```
function addNumbers(a, b) {
  return a + b;
  console.log('Это сообщение не будет выведено в консоль');
}
```

В этом примере сообщение console.log никогда не будет выведено, потому что это находится после оператора return.



Область видимости

Область видимости (scope) в JavaScript определяет доступность переменных, функций и объектов в определенной части кода. Она определяет контекст, в котором переменные могут быть объявлены и использованы.
В JavaScript существует два типа области видимости: глобальная область видимости и локальная область видимости.

Область видимости

Глобальная область видимости: Глобальная область видимости охватывает всю программу JavaScript. Переменные, объявленные в глобальной области видимости, доступны во всем коде.

```
var globalVariable = 'Глобальная переменная';

function globalFunction() {
  console.log(globalVariable); // Доступ к глобальной переменной
}

globalFunction();
```

Область видимости

Локальная область видимости (функциональная область видимости): Переменная, объявленная внутри функции, имеет локальную область видимости. Она доступна только внутри функции, где была объявлена. Это помогает избежать конфликтов имен и сохранить чистоту кода.

```
function localFunction() {
  var localVariable = 'Локальная переменная';
  console.log(localVariable); // Доступ к локальной переменной
}

localFunction();
console.log(localVariable); // Ошибка: localVariable не определена
```

Область видимости

Блочная область видимости: С введением ES6 и ключевых слов let и const в JavaScript появилась концепция блочной области видимости. Переменные, объявленные с помощью let или const, ограничены областью видимости того блока, в котором они были объявлены, а также любыми содержащими блоками.

```
if (true) {
  let blockVar = "Я переменная блочной области видимости";
  console.log(blockVar); // "Я переменная блочной области видимости"
}

console.log(blockVar); // ReferenceError: blockVar is not defined
```

Область видимости

Лексическая область видимости, также известная как статическая область видимости, относится к области видимости, которая определяется структурой и расположением вашего кода, а не порядком вызова функций во время выполнения.
Это означает, что область видимости переменной определяется местом, где эта переменная была объявлена в исходном коде.
Например, переменные, объявленные внутри функции, доступны только внутри этой функции, потому что это определено лексической структурой кода.

В этом примере у вас есть глобальная переменная globalVar, переменная outerVar в области видимости функции outerFunc и переменная innerVar в области видимости функции innerFunc.
Функция innerFunc имеет доступ ко всем переменным, объявленным в ее собственной области видимости, а также ко всем переменным, объявленным в любой из ее родительских областей видимости (в данном случае, outerFunc и глобальная область видимости). Это и есть проявление лексической (статической) области видимости.
Таким образом, лексическая область видимости в JavaScript позволяет функциям иметь доступ к переменным из внешних областей видимости. Это очень важно для понимания таких концепций, как замыкания (closures).

```
let globalVar = 'Я глобальная переменная';

function outerFunc() {
  let outerVar = 'Я переменная внешней функции';

  function innerFunc() {
    let innerVar = 'Я переменная внутренней функции';

    console.log(globalVar); // 'Я глобальная переменная'
    console.log(outerVar); // 'Я переменная внешней функции'
    console.log(innerVar); // 'Я переменная внутренней функции'
  }

  innerFunc();
}

outerFunc();
```

Function Declaration

Объявление функции начинается с ключевого слова function, за которым следует имя функции, список параметров в круглых скобках и затем тело функции в фигурных скобках.

```
function sayHello() {
  console.log('Привет, мир!');
}
sayHello(); // Выведет в консоль: "Привет, мир!"
```

Одно из ключевых свойств Function Declaration – это hoisting, или всплытие. В JavaScript объявления переменных и функций "всплывают" на верх своей области видимости. Это означает, что вы можете вызывать функцию, объявленную с помощью Function Declaration, до того, как она была объявлена в коде.

```
sayHello(); // Выводит 'Привет!' без ошибок

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

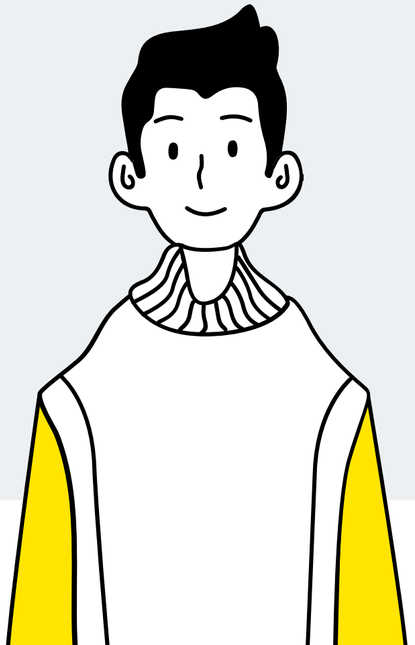
Function Expression

Функциональное выражение – это когда функция объявляется как часть выражения. Это может быть анонимной функцией или именованной функцией, которая сохраняется в переменную.

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

Функции, объявленные через функциональные выражения, не всплывают. Это означает, что если вы попытаетесь вызвать функцию до ее объявления, вы получите ошибку.

```
sayHello(); // Ошибка: sayHello is not defined

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

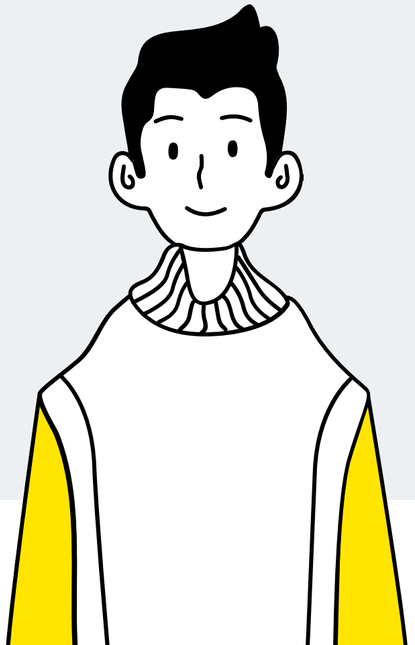


Когда что использовать?

- Function Declarations являются простым способом создания функций, которые можно вызывать из любого места в коде. Они подходят для основных задач и функций уровня приложения.
- Function Expressions обычно используются, когда функция нужна "на лету", например, при передаче функции в качестве аргумента или когда функция используется внутри другого выражения.

Callback

Callback (обратный вызов) – это функция, которая передается в другую функцию в качестве аргумента и будет вызвана позднее внутри этой функции. Когда определенное условие выполняется или определенное действие завершается, callback-функция вызывается для выполнения дополнительных действий или обработки результатов.



Callback

В этом примере функция doSomething принимает callbackFunction в качестве аргумента. Когда doSomething вызывается, она также вызывает переданную callbackFunction, что приводит к выводу сообщения "Выполнился callback!" в консоль. Таким образом, callback-функции позволяют передавать функции в другие функции и вызывать их по требованию, открывая возможности для динамической обработки данных и выполнения дополнительных действий в нужные моменты кода.

```
function doSomething(callback) {
  // Выполняем операции

  // Вызываем callback-функцию
  callback();
}

function callbackFunction() {
  console.log('Выполнился callback!');
}

doSomething(callbackFunction); // Передаем callbackFunction в качестве callback
```

Callback

```
function checkNumber(number, callback) {
  if (number % 2 === 0) {
    callback(true); // Вызываем callback с аргументом true, если число четное
  } else {
    callback(false); // Вызываем callback с аргументом false, если число нечетное
  }
}

function handleResult(isEven) {
  if (isEven) {
    console.log('Число четное');
  } else {
    console.log('Число нечетное');
  }
}

var number = 7;
checkNumber(number, handleResult); // Передаем handleResult как callback
```

В этом примере у нас есть функция checkNumber, которая принимает число и callback-функцию в качестве аргументов. Функция проверяет, является ли число четным или нечетным, и вызывает соответствующий callback с аргументом true или false. Функция handleResult является callback-функцией, которая принимает аргумент isEven и выводит сообщение в зависимости от значения.

При вызове checkNumber(number, handleResult), мы передаем handleResult в качестве callback-функции. Функция checkNumber проверяет число 7 и вызывает handleResult с аргументом false, так как число нечетное. В результате в консоль будет выведено сообщение "Число нечетное".

Таким образом, callback-функции могут использоваться для ветвления и выполнения различных действий в зависимости от результата условия или проверки.

Стрелочные функции

Стрелочные функции (arrow functions) в JavaScript представляют собой синтаксическое сокращение для создания функций. Они были введены в стандарте ECMAScript 6 (ES6) и стали популярным инструментом в веб-разработке благодаря своей краткости и некоторым особенностям.

Стрелочные функции

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

// Стрелочная функция
const add = (a, b) => a + b;
```

Как видите, стрелочная функция гораздо более компактна. Когда функция содержит только одно выражение, можно опустить фигурные скобки и явно указать оператор return. Это называется неявным возвратом (implicit return).
Еще одним сокращением в стрелочных функциях является опускание ключевого слова function, что делает их более лаконичными.

Стрелочные функции

Стрелочные функции особенно полезны в контексте обратного вызова (callback) или при работе с функциями высшего порядка, такими как map, filter и reduce.

```
const numbers = [1, 2, 3, 4, 5];

// Использование обычной функции
const doubled = numbers.map(function(number : number ) {
  return number * 2;
});

// Использование стрелочной функции
const doubled = numbers.map(number => number * 2);
```

Стрелочные функции

Как вы видите, использование стрелочной функции сокращает синтаксис и делает код более читаемым.
Однако важно отметить, что стрелочные функции не подходят для всех случаев. Они не могут быть использованы как конструкторы объектов и не имеют своего собственного arguments или super. Кроме того, из-за отсутствия собственного this, нельзя использовать стрелочные функции для определения методов объекта.
В целом, стрелочные функции – это мощный и удобный инструмент в JavaScript, который упрощает написание кода и сокращает его размер, особенно в контексте функций высшего порядка и обратного вызова.

Стрелочные функции

Основная отличительная черта стрелочных функций состоит в том, что они не создают собственный контекст выполнения (this). Вместо этого, контекст берется из окружающей области видимости (lexical scope) функции. Это означает, что внутри стрелочной функции ключевое слово this ссылается на this внешней функции или блока, в котором она определена.

Thank you!