Шпаргалка: основные конструкции

Условие if

```
if (условие) {
  <сценарий>
}
```

```
if (x % 2 == 0) { // проверка, что число чётное
    alert('Число чётное'); // если утверждение истинно, в окне выводится этот текст
}
```

Условие if else

```
if (условие) {
    <сценарий, если условие выполнено>
} else {
    <сценарий, если условие не выполнено>
}
```

```
if (x % 2 == 0) { // проверка, что число чётное
    alert('Число чётное'); // если чётное, то это выводится в окно
} else {
    alert('Число нечётное'); // если нечётное, то это выводится в окно
}
```

Условие if — else if — else

```
if (условие 1) {
    <сценарий, если условие 1 выполнено>
} else if (условие 2) {
    <сценарий, если условие 1 не выполнено, а условие 2 выполнено>
} else {
    <сценарий, если все условия выше не выполнены>
}
```

```
if (x % 3 == 0) {
    alert('Остаток от деления равен 0');
} else if (x % 3 == 1) {
    alert('Остаток от деления равен 1');
} else {
    alert('Остаток от деления равен 2');
}
```

Условие switch-case

Оператор `switch` нужен, если много похожих друг на друга условий нужно сравнить.

Описание каждого варианта следует за ключевым словом `case` и заканчивается оператором `break`:

```
switch (<переменная для проверки>) {
    case <значение 1>: // if переменная === значение 1
        <выполняемый код> // выполнить код
        break; // завершить выполнение switch
    case <значение 2>: // if переменная === значение 2
        <выполняемый код>
        break;
    ...
    case <значение n>: // if переменная === значение n
       <выполняемый код>
       break;
    default: // if нет подходящего case
       <выполняемый код>
}
```

```
let period = prompt('Выбери период дня');
let periodNormalized = Number(period);

switch (periodNormalized) {
    case 1:
        alert('Скорость 45 км/ч');
        break;
    case 2:
        alert('Скорость 30 км/ч');
        break;
    case 3:
        alert('Скорость 40 км/ч');
        break;
    case 4:
        alert('Скорость 25 км/ч');
        break;
    case 5:
        alert('Скорость 45 км/ч');
        break;
};
```

Цикл for

Цикл обозначают как `for`. Он повторяет действия для определённой переменной.

```
for (<переменная>; <условие>; <шаг>) {
    <сценарий>;
}
```

Этот код интерпретируется так: пока выполняется `условие` для указанной `переменной`, исполняется `сценарий` и делается `шаг`. Шагом может быть, например, инкремент — операция, увеличивающая значение переменной.

```
let sum = 0;

for (let i = 0; i < 5; i++) {
    userNumber = Number(prompt('Введите число'));
    sum = sum + userNumber;
};

alert(sum); // вывод суммы пользователю
```

Сценарий внутри цикла

Например, пользователь вводит числа, которые надо сложить. Но теперь, если пользователь ввёл число больше 10, в сумму нужно добавить 10. Во всех остальных случаях — по-старому: нужно просто складывать числа.

```
let sum = 0;

for (let i = 0; i < 5; i++) {
    userNumber = Number(prompt('Введите число'));
    if (userNumber <= 10) { // проверить число
        sum = sum + userNumber; // если меньше или равно 10, добавляем число к сумме
    } else {
        sum = sum + 10; // если больше 10, добавляем к сумме 10
    }
};

alert(sum);
```

Оператор Continue

Команда `continue` останавливает текущее воспроизведение цикла и сразу начинает следующее.

Например, если нужно вывести в консоль все високосные года с 1890 до 1910, можно применить цикл. Не забудь, каждый сотый год считается невисокосным.

```
for (let i = 1890; i < 1910; i++) {
    if (i % 100 === 0) continue; // если условие выполняется, сразу происходит переход к следующей итерации цикла
    if (i % 4 === 0) {
        console.log('Високосный год: ' + i);
    }
}
```

Оператор Break

Команду `break` прерывает выполнение цикла.

Например, тебе нужно вывести ближайший високосный год в диапазоне от 1601 до 2600 года — это очень много дат. Чтобы вывести високосный год всего один раз, выполнение цикла останавливают сразу после вывода.

```
for (let i = 1601; i < 2600; i++) {
    // Каждый 400-й год опять считается високосным, хотя он делится на 100,
    // поэтому пришлось добавить ещё одно условие.
    if (i % 100 === 0 && i % 400 !== 0) continue;
    if (i % 4 === 0) {
        console.log('Високосный год: ' + i);
        break; // после этой команды выполнение цикла завершится
    }
}
```

Цикл while

`while` — цикл, где не нужно указывать количество повторов, которые надо провести.

```
while (<условие>) {
    <операторы>
};
```

Например, нужно спрашивать имя пользователя до тех пор, пока он его не введёт.

```
let myName;

while (!myName) { // так как myName === undefined, !myName === true.
    myName = prompt('Введите ваше имя: ');
}
```

Если пользователь нажмёт кнопку «Отмена» или оставит поле пустым, программа будет переспрашивать его имя бесконечное число раз.

Например, тебе нужно найти, на каком члене арифметической прогрессии достигается заданное число. Искомое число тебе неизвестно, поэтому нужен цикл без ограничений:

```
let setNumber = Number(prompt('Введите число от 1 до 1000000'));
let summa = 0;
let i = 0;

while (setNumber > summa) { // условие — сравнение исходного числа и суммы
    i = i + 1; // определение следующего числа, которое будет добавлено
    summa = summa + i; // добавление последовательно чисел к сумме
};

// Вывод последнего добавленного числа;
// это и будет нужный член арифметической прогрессии.
alert(i);
```

Цикл do-while

У цикла `while` есть особенность: если условие изначально false, цикл не выполнится ни разу.

Чтобы решить эту проблему, применяют конструкцию `do` `while`: так содержимое цикла выполнится хотя бы один раз. В коде это выглядит так:

```
do {
    <код>
} while (<условие>);
```

`do` `while` — проверка произойдёт, только когда условие выполнится. Значит, пользователя спросят хотя бы один раз.

```
let myName = 'Джон Сноу';
let userName;

do {
    userName = prompt('Введите ваше имя: ');
    if (userName) {
        myName = userName;
    }
} while (!myName);

alert(myName);
```

Функции

Функции: код можно написать один раз, а потом вызывать там, где он нужен.

```
function <название функции>() {
    <код>
}
```

```
function helloFunction() {
    alert('Привет, я функция!');
    alert('Меня можно вызывать из любого места в коде');
    alert('А главное - это можно делать сколько угодно раз');
}

helloFunction();
helloFunction();
helloFunction();
helloFunction();
```

Есть ещё одна форма записи — стрелочная функция: применяется оператор стрелка `=>`. Сравни два варианта одной и той же функции:

```
// классическая функция
function sum(a, b){
  console.log(a + b);
}

// стрелочная функция
let sum = (a, b) => console.log(a + b);
```

Параметры и аргументы

Чтобы не указывать конкретные значения в функции каждый раз, можно описать **параметры**, или входные значения функции. Параметры указывают в круглых скобках `( )`, когда объявляют

функцию.

```
function <название функции> (<параметр_1>, <параметр_2>, ..., <параметр_N>) {
    <код>
}
```

```
// объявление функции
function hello(text) {
// обращение к параметру, как будто это переменная
    console.log('Hello, ' + text);
}
```

Когда вызывают функцию, указывают **аргументы** — значения параметров. Среди них могут быть и определённое значение, и переменная, и выражение.

Например, ты заказываешь еду в кафе. Официант — функция. Он знает, что примет список блюд — список параметров функции. Когда ты называешь блюда, ты передаёшь официанту аргументы — значения параметров.

```
function <название функции> (<параметр_1>, <параметр_2>, ..., <параметр_N>) {
    <код>
}
// вызов функции
<название функции> (<аргумент_1>, <аргумент_2>,.., <аргумент_M>)
```

```
// объявление функции
function hello(text) {
// обращение к параметру, как будто это переменная
    console.log('Hello, ' + text);
}

// Пример 1
let person = 'Вася';
// вызов функции с аргументом person
hello(person); // внутри функции параметр text = person

// Пример 2
let oldMan = 'Дедушка';
// вызов функции с аргументом oldMan
hello(oldMan); // внутри функции параметр text = oldMan

// Пример 3
// вызов функции с аргументом 'world'
hello('world'); // внутри функции параметр text = 'world'
```

Аргументов и параметров может быть несколько.

```
// создаётся функция с двумя параметрами
function math(num1, num2) {
    console.log('Умножение: ' + (num1 * num2));
    console.log('Сложение: ' + (num1 + num2));
}

// вызов функции с нужными аргументами
math(3, 3); // выведет 9 для умножения и 6 для сложения
math(4, 5); // выведет 20 для умножения и 9 для сложения
```

Если аргументов больше, чем параметров, лишние аргументы не присвоятся ни одному параметру. В следующем примере функция вызывается с четырьмя аргументами вместо двух. Значения 4 и 5 не будут задействованы.

```
// у функции два параметра
function math(num1, num2) {
    console.log('Умножение: ' + (num1 * num2)); // выведет «Умножение: 9»
    console.log('Сложение: ' + (num1 + num2)); // выведет «Сложение: 6»
}

math(3, 3, 4, 5); // но вызов с 4 аргументами (4 аргумента > 2 параметра)
```

Если у функции больше параметров, чем аргументов, параметрам без аргументов присваивается значение `undefined`. В этом примере `num2 = undefined`. Функция вернёт значение NaN — специальное обозначение Not a number (англ. «Не число» или «Неопределённое число»). Его применяют, чтобы обозначить ошибку при математических операциях.

```
// у функции два параметра
function math(num1, num2) {
    console.log('Умножение: ' + (num1 * num2));
    console.log('Сложение: ' + (num1 + num2));
}

math(3); // но вызов с 1 аргументом
```

Оператор return

Функция возвращает результат работы по команде `return`. Он вернётся в место в коде, откуда вызвали функцию. Как команда выглядит в коде:

```
return <выражение>;
```

Команду `return` можно поставить в любое место в функции. Когда посчитается `return`, функция вернёт значение и завершит выполнение. Всё, что идёт после команды `return`, программа игнорирует.

```
function math(num1, num2) {
    return (num1 * num2); // функция завершает работу
    alert('Здесь умножаем'); // не будет выполнено
}

// Вызов функции с нужными аргументами
let result = math(3, 4); // присвоит переменной result значение 12.
```

Чтобы применить результат функции, его можно присвоить какой-нибудь переменной, например `x`:

```
// создаётся функция с двумя параметрами
function math(num1, num2) { // num1 = 3, num 2 = 4
    let x = num1 * num2; // x = 12
    return x; // вернёт 12 в место вызова функции
}

// Вызов функции с нужными аргументами
let result = math(3, 4); // присвоит переменной result значение 12.
```

Внутри функции может быть несколько команд `return`. Функция завершит работу, как только посчитается один `return`.

Функция завершит работу после первой команды `return`. Иначе — значения складываются, и работа завершается на втором `return`.

```
function math(num1, num2) {
    if num1 > num2 {
        return num1 * num2; // первый return
    } else {
        return num1 + num2; // второй return
    }
}

// вызов функции с нужными аргументами
let result = math(4, 3); // функция вернёт 12
let result = math(3, 4); // функция вернёт 7
```

Функция возвращает значение `undefined` в двух случаях.

Первый — если не указана команда `return`. Тогда функция становится **процедурой** — это функции, которые не отдают результат.

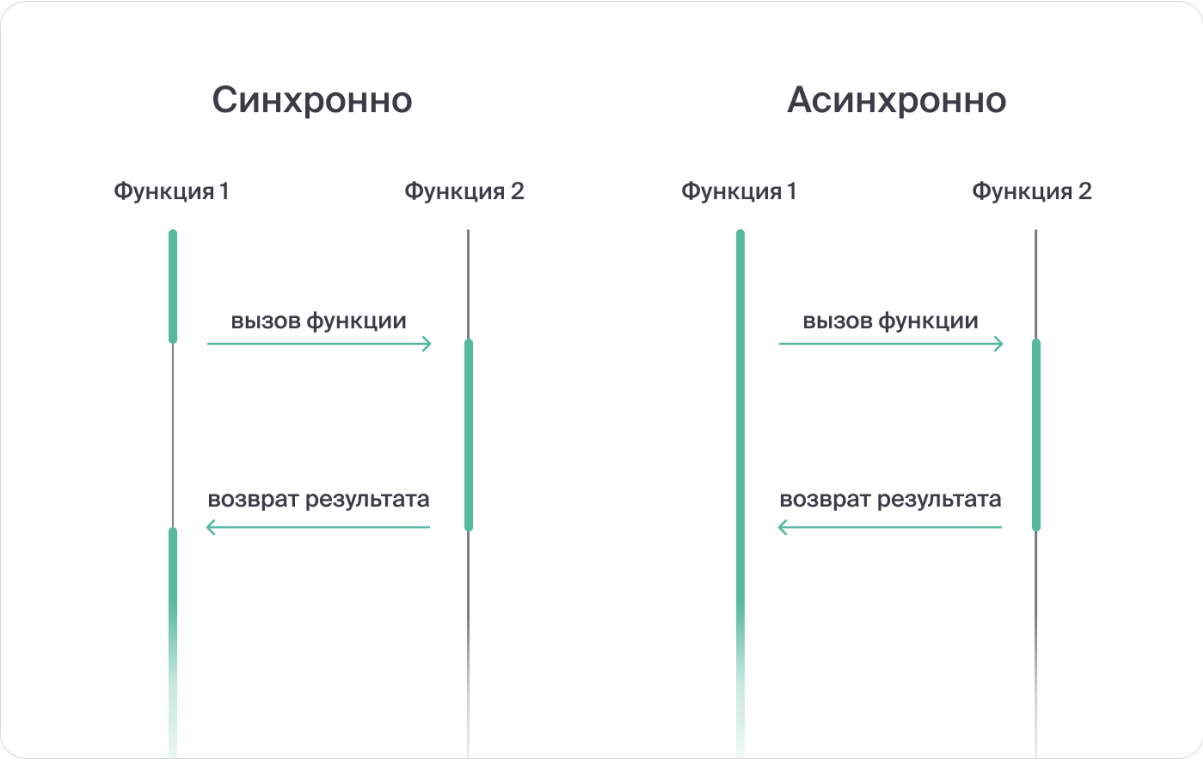
```
function math(num1, num2) {
    alert(num1 * num2); // нет return
};

let result = math(3, 4); // присвоит undefined переменной result
```

Второй случай — если не указано значение, которое должно вернуться. Например, не указали, что должен вернуться результат `x`. Команда `return` завершит работу функции. Код `return` проигнорируется, функция вернёт `undefined`.

```
function math(num1, num2) {
    let x = num1 * num2;
    return; // не указано возвращаемое значение
    alert('Здесь умножают'); // игнорируется
};
// Вызов функции с нужными аргументами
let result = math(3, 4); // присвоит undefined переменной result.
```

Синхронность и асинхронность

Как асинхронный подход отличается от синхронного:



Синхронность

Программа ждёт выполнения очередной операции, хотя может параллельно запускать и другие. К примеру, объявляют функцию `load`. Она эмулирует загрузку файла: предупреждает о начале загрузки файла, загружает его и сообщает об окончании загрузки.

```
function load(){
    console.log('Начало загрузки');
    progress();
    console.log('Загрузка окончена');
}

function progress(){
    emulation(100000000);
    console.log('Загружено');
}

function emulation(n){
    for(let i = 0; i < n; i++){
    }
}

load(); //вызов функции загрузки
```

В коде функции `load` вызывается функция `progress`. Она загружает файл и сообщает, что файл загружен.

Внутри функции `progress` вызывается функция `emulation`. Эта функция должна загружать файл, но тогда пример займёт много места: в примере реализации кода применяют цикл.

Асинхронность

Асинхронный подход — это противоположность синхронности. Чтобы функция не ждала выполнения остальных команд — была асинхронной — примени ключевое слово `async`.

Асинхронной функции нужно явно указать, что она ждёт завершения какой-то команды. Это делают командой `await`. Её можно применять только внутри асинхронной функции. Как выглядит код теперь:

```
async function load(){
    console.log('Начало загрузки');
    progress();
    console.log('Загрузка окончена');
}

async function progress(){
    await emulation(100000000);
    console.log('Загружено');
}

function emulation(n){
    for(let i = 0; i < n; i++){
    }
}

load();
```

Вызов функции `emulation` помечен `await`. В этом месте функция `progress` будет ждать выполнения команды. При этом функция `progress` асинхронная: она отправит сообщение вызвавшей её функции `load` со смыслом «не жди меня, я тут надолго застряла».