Содержание:

Операторы нулевого слияния и присваивания: '??', '??='

Тернарный оператор

Циклы while и for

Конструкция "switch"

Операторы нулевого слияния и

присваивания: '??', '??='

Оператор нулевого слияния (??)

Оператор нулевого слияния представляет собой два вопросительных знака ??.

Так как он обрабатывает null и undefined одинаковым образом, то для этой статьи

мы введём специальный термин. Для краткости будем говорить, что значение

«определено», если оно не равняется ни null, ни undefined.

Результат выражения a ?? b будет следующим:

если a определено, то a,

если a не определено, то b.

Иначе говоря, оператор ?? возвращает первый аргумент, если он не null/undefined,

иначе второй.

Оператор нулевого слияния не является чем-то принципиально новым. Это всего

лишь удобный синтаксис, как из двух значений получить одно, которое

«определено».

Вот как можно переписать выражение result = a ?? b, используя уже знакомые

нам операторы:

result = (a !== null && a !== undefined) ? a : b;

Теперь должно быть абсолютно ясно, что делает ??. Давайте посмотрим, где это

может быть полезно.

Как правило, оператор ?? нужен для того, чтобы задать значение по умолчанию

для потенциально неопределённой переменной.

Например, здесь мы отобразим user, если её значение не null/undefined, в

противном случае Аноним:

let user;

alert(user ?? "Аноним"); // Аноним (user не существует)

А вот пример, когда user присвоено значение:

let user = "Иван";

alert(user ?? "Аноним"); // Иван (user существует)

Кроме этого, можно записать последовательность из операторов ??, чтобы

получить первое значение из списка, которое не является null/undefined.

Допустим, у нас есть данные пользователя в

переменных firstName, lastName или nickName. Все они могут не существовать, если

пользователь решил не вводить соответствующие значение.

Мы хотели бы отобразить имя пользователя, используя одну из этих переменных,

или показать «Аноним», если все они null/undefined.

Для этого воспользуемся оператором ??:

let firstName = null;

let lastName = null;

let nickName = "Суперкодер";

// показывает первое значение, которое определено:

alert(firstName ?? lastName ?? nickName ?? "Аноним"); // Суперкодер

Сравнение с ||

Оператор ИЛИ || можно использовать для того же, что и ??, как это было показано

в предыдущем уроке.

Например, если в приведённом выше коде заменить ?? на ||, то будет тот же

самый результат:

let firstName = null;

let lastName = null;

let nickName = "Суперкодер";

// показывает первое истинное значение:

alert(firstName || lastName || nickName || "Аноним"); // Суперкодер

Исторически сложилось так, что оператор ИЛИ || появился первым. Он существует

с самого начала в JavaScript, поэтому разработчики долгое время использовали его

для таких целей.

С другой стороны, сравнительно недавно в язык был добавлен оператор нулевого

слияния ?? – как раз потому, что многие были недовольны оператором ||.

Важное различие между ними заключается в том, что:

|| возвращает первое истинное значение.

?? возвращает первое определённое значение.

Проще говоря, оператор || не различает false, 0, пустую строку "" и null/undefined.

Для него они все одинаковы, т.е. являются ложными значениями. Если первым

аргументом для оператора || будет любое из перечисленных значений, то в

качестве результата мы получим второй аргумент.

Однако на практике часто требуется использовать значение по умолчанию только

тогда, когда переменная является null/undefined. Ведь именно тогда значение

действительно неизвестно/не определено.

Рассмотрим следующий пример:

let height = 0;

alert(height || 100); // 100

alert(height ?? 100); // 0

height || 100 проверяет height на ложное значение, оно равно 0, да, ложное.

o поэтому результатом || является второй аргумент, т.е. 100.

height ?? 100 проверяет, что переменная height содержит null/undefined, а

поскольку это не так,

o то результатом является сама переменная height, т.е. 0.

На практике нулевая высота часто является вполне нормальным значением,

которое не следует заменять значением по умолчанию. Таким образом, ?? здесь

как раз работает так, как нужно.

Приоритет

Приоритет оператора ?? такой же, как и у ||. Они оба равны 3 в таблице на MDN.

Это означает, что, как и ||, оператор нулевого слияния ?? вычисляется до = и ?, но

после большинства других операций, таких как +, \*.

Так что, в выражениях такого вида понадобятся скобки:

let height = null;

let width = null;

// важно: используйте круглые скобки

let area = (height ?? 100) \* (width ?? 50);

alert(area); // 5000

Иначе, если опустить скобки, оператор \* выполнится первым, так как у него

приоритет выше, чем у ??, и это приведёт к неправильным результатам.

// без скобок

let area = height ?? 100 \* width ?? 50;

// ...сработает вот так (совсем не как нам нужно):

let area = height ?? (100 \* width) ?? 50;

Использование ?? вместе с && или ||

По соображениям безопасности JavaScript запрещает использование

оператора ?? вместе с && и ||, если приоритет явно не указан при помощи круглых

скобок.

Выполнение следующего кода приведёт к синтаксической ошибке:

let x = 1 && 2 ?? 3; // Синтаксическая ошибка

Это, безусловно, спорное ограничение было добавлено в спецификацию языка с

целью избежать программные ошибки, когда люди начнут переходить с || на ??.

Используйте скобки, чтобы обойти это ограничение:

let x = (1 && 2) ?? 3; // Работает без ошибок

alert(x); // 2

Оператор нулевого присваивания (??=)

Предположим, нам необходимо проверить, равна ли

переменная null или undefined, и если это так — присвоить этой переменной какое-

либо другое значение.

Вот как мы сделали бы это сейчас:

let userAge = null;

if (userAge === null || userAge === undefined) {

userAge = 18;

}

Выглядит громоздко, правда? Существует оператор, более подходящий для

подобных задач. Вот его синтаксис:

x ??= y

Оператор ??= присвоит x значение y только в том случае, если x не

определено (null/undefined).

Теперь попробуем переписать уже знакомый нам фрагмент кода используя новый

оператор:

let userAge = null;

userAge ??= 18;

alert(userAge) // 18

Обратите внимание: если бы userAge не был равен null/undefined, то выражение

справа от ??= никогда бы не выполнилось:

let userAge = 18;

userAge ??= alert("не сработает");

userAge ??= 21;

userAge ??= null;

alert(userAge) // по-прежнему 18

Итого

Оператор нулевого слияния ?? — это быстрый способ выбрать первое

«определённое» значение из списка.

Используется для присвоения переменным значений по умолчанию:

// будет height=100, если переменная height равна null или undefined

height = height ?? 100;

Оператор ?? имеет очень низкий приоритет, лишь немного выше, чем у ? и =,

поэтому при использовании его в выражении, скорее всего, потребуются скобки.

Запрещено использовать вместе с || или && без явно указанного приоритета, то

есть без скобок.

Для присвоения переменной значения в зависимости от того, «определена» она

или нет, используется оператор нулевого присваивания ??=.

Тернарный оператор

Посмотрите на определение функции, которая возвращает модуль переданного

числа:

const abs = (number) => {

if (number >= 0) {

return number;

}

return -number;

};

abs(10); // 10

abs(-10); // 10

Можно ли записать её лаконичнее? Что-то вроде return <ответ в зависимости от

условия>? Для этого справа от return должно быть выражение, но if — это

инструкция, а не выражение.

В JavaScript существует конструкция, которая по своему действию аналогична

конструкции if-else, но при этом является выражением. Она называется тернарный

оператор.

Тернарный оператор — единственный в своем роде оператор, требующий три

операнда:

const abs = (number) => {

return number >= 0 ? number : -number;

};

Общий паттерн выглядит так: <predicate> ? <expression on true> : <expression on false>.

Сокращенный вариант функции abs(), выглядит так:

const abs = (number) => (number >= 0 ? number : -number);

Обратите внимание на скобки вокруг тернарника. Они не обязательны, но

линтер настоятельно рекомендует их ставить, во избежание неоднозначностей.

Давайте перепишем начальный вариант getTypeOfSentence() аналогично:

Было:

const getTypeOfSentence = (sentence) => {

const lastChar = sentence.slice(-1);

if (lastChar === '?') {

return 'question';

}

return 'normal';

};

Стало:

const getTypeOfSentence = (sentence) => {

const lastChar = sentence.slice(-1);

return (lastChar === '?') ? 'question' : 'normal';

};

getTypeOfSentence('Hodor'); // normal

getTypeOfSentence('Hodor?'); // question

Если вы помните, в чём сила выражений, то вероятно уже догадались, что

тернарный оператор можно вкладывать в тернарный оператор. Не делайте этого

:) Такой код тяжело и читать, и отлаживать, это очень плохая практика.

Циклы while и for

При написании скриптов зачастую встаёт задача сделать однотипное действие

много раз.

Например, вывести товары из списка один за другим. Или просто перебрать все

числа от 1 до 10 и для каждого выполнить одинаковый код.

Для многократного повторения одного участка кода предусмотрены циклы.

Циклы for...of и for...in

Цикл «while»

Цикл while имеет следующий синтаксис:

while (condition) {

// код

// также называемый "телом цикла"

}

Код из тела цикла выполняется, пока условие condition истинно.

Например, цикл ниже выводит i, пока i < 3:

let i = 0;

while (i < 3) { // выводит 0, затем 1, затем 2

alert( i );

i++;

}

Одно выполнение тела цикла по-научному называется итерация. Цикл в примере

выше совершает три итерации.

Если бы строка i++ отсутствовала в примере выше, то цикл бы повторялся (в

теории) вечно. На практике, конечно, браузер не позволит такому случиться, он

предоставит пользователю возможность остановить «подвисший» скрипт, а

JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только

сравнение: условие while вычисляется и преобразуется в логическое значение.

Например, while (i) – более краткий вариант while (i != 0):

let i = 3;

while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится

alert( i );

i--;

}

Фигурные скобки не требуются для тела цикла из одной строки

Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные

скобки {...}:

let i = 3;

while (i) alert(i--);

Цикл «do...while»

Проверку условия можно разместить под телом цикла, используя специальный

синтаксис do..while:

do {

// тело цикла

} while (condition);

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его

значение равно true, он будет выполняться снова и снова.

Например:

let i = 0;

do {

alert( i );

i++;

} while (i < 3);

Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла

выполнилось хотя бы один раз, даже если условие окажется ложным. На практике

чаще используется форма с предусловием: while(...) {...}.

Цикл «for»

Более сложный, но при этом самый распространённый цикл — цикл for.

Выглядит он так:

for (начало; условие; шаг) {

// ... тело цикла ...

}

Давайте разберёмся, что означает каждая часть, на примере. Цикл ниже

выполняет alert(i) для i от 0 до (но не включая) 3:

for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2

alert(i);

}

Рассмотрим конструкцию for подробней:

часть

начало let i = 0 Выполняется один раз при входе в цикл

условие i < 3 Проверяется перед каждой итерацией цикла.

Если оно вычислится в false, цикл остановится.

тело alert(i) Выполняется снова и снова, пока условие вычисляется в true.

шаг i++ Выполняется после тела цикла на каждой итерации перед проверкой условия.

В целом, алгоритм работы цикла выглядит следующим образом:

Выполнить начало

→ (Если условие == true → Выполнить тело, Выполнить шаг)

→ (Если условие == true → Выполнить тело, Выполнить шаг)

→ (Если условие == true → Выполнить тело, Выполнить шаг)

→ ...

То есть, начало выполняется один раз, а затем каждая итерация заключается в

проверке условия, после которой выполняется тело и шаг.

Если тема циклов для вас нова, может быть полезным вернуться к примеру выше и

воспроизвести его работу на листе бумаги, шаг за шагом.

Вот в точности то, что происходит в нашем случае:

// for (let i = 0; i < 3; i++) alert(i)

// Выполнить начало

let i = 0;

// Если условие == true → Выполнить тело, Выполнить шаг

if (i < 3) { alert(i); i++ }

// Если условие == true → Выполнить тело, Выполнить шаг

if (i < 3) { alert(i); i++ }

// Если условие == true → Выполнить тело, Выполнить шаг

if (i < 3) { alert(i); i++ }

// ...конец, потому что теперь i == 3

Встроенное объявление переменной

В примере переменная счётчика i была объявлена прямо в цикле. Это так

называемое «встроенное» объявление переменной. Такие переменные

существуют только внутри цикла.

for (let i = 0; i < 3; i++) {

alert(i); // 0, 1, 2

}

alert(i); // ошибка, нет такой переменной

Вместо объявления новой переменной мы можем использовать уже

существующую:

let i = 0;

for (i = 0; i < 3; i++) { // используем существующую переменную

alert(i); // 0, 1, 2

}

alert(i); // 3, переменная доступна, т.к. была объявлена снаружи цикла

Пропуск частей «for»

Любая часть for может быть пропущена.

Для примера, мы можем пропустить начало если нам ничего не нужно делать перед

стартом цикла.

Вот так:

let i = 0; // мы уже имеем объявленную i с присвоенным значением

for (; i < 3; i++) { // нет необходимости в "начале"

alert( i ); // 0, 1, 2

}

Можно убрать и шаг:

let i = 0;

for (; i < 3;) {

alert( i++ );

}

Это сделает цикл аналогичным while (i < 3).

А можно и вообще убрать всё, получив бесконечный цикл:

for (;;) {

// будет выполняться вечно

}

При этом сами точки с запятой ; обязательно должны присутствовать, иначе будет

ошибка синтаксиса.

Прерывание цикла: «break»

Обычно цикл завершается при вычислении условия в false.

Но мы можем выйти из цикла в любой момент с помощью специальной

директивы break.

Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока

посетитель их вводит, а затем – выдаёт:

let sum = 0;

while (true) {

let value = +prompt("Введите число", '');

if (!value) break; // (\*)

sum += value;

}

alert( 'Сумма: ' + sum );

Директива break в строке (\*) полностью прекращает выполнение цикла и передаёт

управление на строку за его телом, то есть на alert.

Вообще, сочетание «бесконечный цикл + break» – отличная штука для тех ситуаций,

когда условие, по которому нужно прерваться, находится не в начале или конце

цикла, а посередине или даже в нескольких местах его тела.

Переход к следующей итерации: continue

Директива continue – «облегчённая версия» break. При её выполнении цикл не

прерывается, а переходит к следующей итерации (если условие все ещё

равно true).

Её используют, если понятно, что на текущем повторе цикла делать больше нечего.

Например, цикл ниже использует continue, чтобы выводить только нечётные

значения:

for (let i = 0; i < 10; i++) {

// если true, пропустить оставшуюся часть тела цикла

if (i % 2 == 0) continue;

alert(i); // 1, затем 3, 5, 7, 9

}

Для чётных значений i, директива continue прекращает выполнение тела цикла и

передаёт управление на следующую итерацию for (со следующим числом). Таким

образом alert вызывается только для нечётных значений.

Директива continue позволяет избегать вложенности

Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

for (let i = 0; i < 10; i++) {

if (i % 2) {

alert( i );

}

}

С технической точки зрения он полностью идентичен. Действительно,

вместо continue можно просто завернуть действия в блок if.

Однако мы получили дополнительный уровень вложенности фигурных скобок.

Если код внутри if более длинный, то это ухудшает читаемость, в отличие от

варианта с continue.

Нельзя использовать break/continue справа от оператора „?“

Обратите внимание, что эти синтаксические конструкции не являются

выражениями и не могут быть использованы с тернарным оператором ?. В

частности, использование таких директив, как break/continue, вызовет ошибку.

Например, если мы возьмём этот код:

if (i > 5) {

alert(i);

} else {

continue;

}

...и перепишем его, используя вопросительный знак:

(i > 5) ? alert(i) : continue; // continue здесь приведёт к ошибке

...то будет синтаксическая ошибка.

Это ещё один повод не использовать оператор вопросительного знака ? вместо if.

Метки для break/continue

Бывает, нужно выйти одновременно из нескольких уровней цикла сразу.

Например, в коде ниже мы проходимся циклами по i и j, запрашивая с

помощью prompt координаты (i, j) с (0,0) до (2,2):

for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`Значение на координатах (${i},${j})`, '');

// Что если мы захотим перейти к Готово (ниже) прямо отсюда?

}

}

alert('Готово!');

Нам нужен способ остановить выполнение, если пользователь отменит ввод.

Обычный break после input лишь прервёт внутренний цикл, но этого недостаточно.

Достичь желаемого поведения можно с помощью меток.

Метка имеет вид идентификатора с двоеточием перед циклом:

labelName: for (...) {

...

}

Вызов break <labelName> в цикле ниже ищет ближайший внешний цикл с такой

меткой и переходит в его конец.

outer: for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`Значение на координатах (${i},${j})`, '');

// если пустая строка или Отмена, то выйти из обоих циклов

if (!input) break outer; // (\*)

// сделать что-нибудь со значениями...

}

}

alert('Готово!');

В примере выше это означает, что вызовом break outer будет разорван внешний

цикл до метки с именем outer.

Таким образом управление перейдёт со строки, помеченной (\*), к alert('Готово!').

Можно размещать метку на отдельной строке:

outer:

for (let i = 0; i < 3; i++) { ... }

Директива continue также может быть использована с меткой. В этом случае

управление перейдёт на следующую итерацию цикла с меткой.

Метки не позволяют «прыгнуть» куда угодно

Метки не дают возможности передавать управление в произвольное место кода.

Например, нет возможности сделать следующее:

break label; // не прыгает к метке ниже

label: for (...)

Директива break должна находиться внутри блока кода. Технически, подойдет

любой маркированный блок кода, например:

label: {

// ...

break label; // работает

// ...

}

...Хотя в 99.9% случаев break используется внутри циклов, как мы видели в

примерах выше.

К слову, continue возможно только внутри цикла.

Итого

Мы рассмотрели 3 вида циклов:

while – Проверяет условие перед каждой итерацией.

do..while – Проверяет условие после каждой итерации.

for (;;) – Проверяет условие перед каждой итерацией, есть возможность задать

дополнительные настройки.

Чтобы организовать бесконечный цикл, используют конструкцию while (true). При

этом он, как и любой другой цикл, может быть прерван директивой break.

Если на данной итерации цикла делать больше ничего не надо, но полностью

прекращать цикл не следует – используют директиву continue.

Обе этих директивы поддерживают метки, которые ставятся перед циклом. Метки

– единственный способ для break/continue выйти за пределы текущего цикла,

повлиять на выполнение внешнего.

Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript

нет такой возможности.

Дополнительно про цикл While

Допустим мы хотим написать функцию, которая выводит на экран все числа от 1 до

указанного (через аргументы):

printNumbers(3);

// => 1

// => 2

// => 3

Эту функцию невозможно реализовать уже изученными средствами, так как

количество выводов на экран заранее неизвестно. А с циклами это не составит

никаких проблем:

const printNumbers = (lastNumber) => {

// i сокращение от index (порядковый номер)

// используется по общему соглашению во множестве языков

// как счетчик цикла

let i = 1;

while (i <= lastNumber) {

console.log(i);

i = i + 1;

}

console.log('finished!');

};

printNumbers(3);

1

2

3

finished!

В коде функции использован цикл while. Он состоит из трёх элементов:

Ключевое слово while. Несмотря на схожесть с вызовом функций, это не

вызов функции.

Предикат. Условие, которое указывается в скобках после while. Это условие

вычисляется и проверяется перед выполнением тела цикла на каждой

итерации.

Тело цикла. Блок кода в фигурных скобках. Этот блок аналогичен блоку кода

в функциях. Всё, что определено внутри этого блока (константы или

переменные), видно только внутри этого блока.

Конструкция читается так: «пока истинно условие (предикат) i <= lastNumber делать то,

что указано в теле цикла». Разберём работу этого кода для вызова printNumbers(3):

// Инициализируется i

let i = 1;

// Предикат возвращает true, поэтому выполняется тело цикла

while (1 <= 3)

// console.log(1);

// i = 1 + 1;

// Закончилось тело цикла, поэтому происходит возврат в начало

while (2 <= 3)

// console.log(2);

// i = 2 + 1;

// Закончилось тело цикла, поэтому происходит возврат в начало

while (3 <= 3)

// console.log(3);

// i = 3 + 1;

// Предикат возвращает false, поэтому выполнение переходит за цикл

while (4 <= 3)

// console.log('finished!');

// На этом этапе i равен 4, но он нам уже не нужен

// функция завершается

Самое главное в цикле — завершение его работы (выход из цикла). Процесс,

который порождает цикл, должен в конце концов остановиться. Ответственность

за остановку полностью лежит на программисте. Обычно задача сводится к

введению переменной, называемой «счётчиком цикла». Сначала счётчик

инициализируется, то есть ему задаётся начальное значение. В нашем примере это

инструкция let i = 1, выполняемая до входа в цикл. Затем в условии цикла

проверяется, достиг ли счётчик своего предельного значения. И, наконец, счётчик

меняет свое значение i = i + 1.

На этом моменте новички делают больше всего ошибок. Например, случайно

забытое увеличение счётчика или неправильная проверка в предикате способны

привести к зацикливанию. Это ситуация, при которой цикл работает бесконечно и

программа никогда не останавливается. В таком случае приходится её завершать

принудительно (кто знает, может быть когда зависают реальные программы, в этот

момент внутри них выполняется бесконечный цикл).

const printNumbers = (lastNumber) => {

let i = 1;

// Этот цикл никогда не остановится

// и будет печатать всегда одно значение

while (i <= lastNumber) {

console.log(i);

}

console.log('finished!');

};

В некоторых случаях бесконечные циклы полезны. Здесь мы такие случаи не

рассматриваем, но полезно видеть как выглядит этот код:

while (true) {

// Что-то делаем

}

Подводя итог. Когда всё же нужны циклы, а когда можно обойтись без них?

Физически невозможно обойтись без циклов тогда, когда алгоритм решения

задачи требует повторения каких-то действий, как в примере с книгой, и

количество этих операций заранее неизвестно.