Лабораторная работа № 2

Тема: *Основы JavaScript.*
Цель: *ознакомиться с основами JavaScript, научиться применять их на практике.*

1 Краткая теория

1.1 ПЕРЕМЕННЫЕ

Переменная – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Для создания переменной в JavaScript используйте ключевое слово **let**.

Приведённая ниже инструкция создаёт (другими словами: *объявляет* или *определяет*) переменную с именем «message»:

```
let message;
```

Теперь можно поместить в неё данные, используя оператор присваивания =:

```
message = 'Hello'; // сохранить строку
```

В старых скриптах вы также можете найти другое ключевое слово: **var** вместо **let**:

```
var message = 'Hello';
```

Ключевое слово var – *почти* то же самое, что и let. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

1.2 ИМЕННА ПЕРЕМЕННЫХ

В JavaScript есть два ограничения, касающиеся имён переменных:

1. Имя переменной должно содержать только буквы, цифры или символы **$** и **_**.
2. Первый символ не должен быть цифрой.

Примеры допустимых имён:

```
let userName;
let test123;
```

Если имя содержит несколько слов, обычно используется верблюжья нотация, то есть, слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: myVeryLongName.

Самое интересное – знак доллара '$' и подчёркивание '_' также можно использовать в названиях. Это обычные символы, как и буквы, без какого-либо особого значения.

Эти имена являются допустимыми:

```
let $ = 1; // объявили переменную с именем "$"
let _ = 2; // а теперь переменную с именем "_"
alert($ + _); // 3
```

! Регистр имеет значение !

Переменные с именами apple и AppLE – это две разные переменные.

! Нелатинские буквы разрешены, но не рекомендуются !

Можно использовать любой язык, включая кириллицу или даже иероглифы, например:

```
let имя = '...';
let 我 = '...';
```

Технически здесь нет ошибки, такие имена разрешены, но есть международная традиция использовать английский язык в именах переменных. Даже если мы пишем небольшой скрипт, у него может быть долгая жизнь впереди. Людям из других стран, возможно, придётся прочесть его не один раз.

! Зарезервированные имена !

Существует список зарезервированных слов, которые нельзя использовать в качестве имён переменных, потому что они используются самим языком.

Например: let, class, return и function зарезервированы.

Приведённый ниже *код даёт синтаксическую ошибку*:

```
let let = 5; // нельзя назвать переменную "let", ошибка!
let return = 5; // также нельзя назвать переменную "return", ошибка!
```

1.3 КОНСТАНТЫ

Чтобы объявить константную, то есть, неизменяемую переменную, используйте **const** вместо let:

```
const myBirthday = '18.04.1982';
```

! Несколько хороших правил: **!**

- Используйте легко читаемые имена, такие как userName или shoppingCart.
- Избегайте использования аббревиатур или коротких имён, таких как a, b, c, за исключением тех случаев, когда вы точно знаете, что так нужно.
- Делайте имена максимально описательными и лаконичными. Примеры плохих имён: data и value. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно, какие данные хранит переменная.
- Договоритесь с вашей командой об используемых терминах. Если посетитель сайта называется «user», тогда мы должны называть связанные с ним переменные currentUser или newUser, а не currentVisitor или newManInTown.

1.4 ТИПЫ ДАННЫХ

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

```
// Не будет ошибкой
let message = "hello";
message = 123456;
```

Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.

! **Числовой *тип*** данных (**number**) представляет как целочисленные значения, так и числа с плавающей точкой.

Существует множество операций для чисел, например, умножение *, деление /, сложение +, вычитание - и так далее.

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinityи NaN.

- Infinity представляет собой математическую бесконечность ∞. Это особое значение, которое больше любого числа.
- Мы можем получить его в результате деления на ноль:

```
alert( 1 / 0 ); // Infinity
```

Или задать его явно:

```
alert( Infinity ); // Infinity
```

- NaN означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

```
alert( "не число" / 2 ); // NaN, такое деление является ошибкой
```

Значение NaN «прилипчиво». Любая операция с NaN возвращает NaN:

```
alert( "не число" / 2 + 5 ); // NaN
```

Если где-то в математическом выражении есть NaN, то результатом вычислений с его участием будет NaN.

! **Тип BigInt** был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.

Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

```
// символ "n" в конце означает, что это BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

! **Строка (string)** в JavaScript должна быть заключена в кавычки.

В JavaScript существует три типа кавычек.

1. Двойные кавычки: "Привет".
2. Одинарные кавычки: 'Привет'.
3. Обратные кавычки: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные же кавычки имеют расширенную функциональность. Они позволяют нам встраивать выражения в строку, заключая их в ${…}.

```
let name = "Иван";
// Вставим переменную
alert( `Привет, ${name}!` ); // Привет, Иван!
// Вставим выражение
alert( `результат: ${1 + 2}` ); // результат: 3
```

! **Булевый тип (boolean)** может принимать только два значения: true(истина) и false (ложь).

Такой тип, как правило, используется для хранения значений да/нет: true значит «да, правильно», а false значит «нет, не правильно».

! **Специальное значение null** не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение null:

```
let age = null;
```

В JavaScript null *не является «ссылкой на несуществующий объект» или «нулевым указателем»*, как в некоторых других языках.

Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

В приведённом выше коде указано, что значение переменной age неизвестно.

! **Специальное значение undefined** формирует тип из самого себя так же, как и null.

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:

```
let age;
alert(age); // выведет "undefined"
```

! **Тип object (объект)** – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то ещё). В объектах же хранят коллекции данных или более сложные структуры.

! **Тип symbol (символ)** используется для создания уникальных идентификаторов в объектах.

Оператор typeof возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.

У него есть две синтаксические формы:

1. Синтаксис оператора: typeof x.
2. Синтаксис функции: typeof(x).

1.5 ВЗАИМОДЕЙСТВИЕ: alert, prompt, confirm

alert – показывает сообщение.

prompt – показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный в поле ввода текст или null, если была нажата кнопка «Отмена» или Esc с клавиатуры.

confirm – показывает сообщение и ждёт, пока пользователь нажмёт OK или Отмена. Возвращает true, если нажата OK, и false, если нажата кнопка «Отмена» или Esc с клавиатуры.

Все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

! *На все указанные методы распространяются два ограничения:* !

1. Расположение окон определяется браузером. Обычно окна находятся в центре.
2. Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.

1.6 ПРЕОБРАЗОВАНИЕ ТИПОВ

Существует 3 наиболее широко используемых преобразования: строковое, численное и логическое.

Строковое – Происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью String(value). Для примитивных значений работает очевидным образом.

Численное – Происходит в математических операциях. Может быть вызвано с помощью Number(value).

! *Преобразование подчиняется правилам*: !

Значение	Становится...
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем `0`, иначе из непустой строки «считывается» число. При ошибке результат `NaN`.

Логическое – Происходит в логических операциях. Может быть вызвано с помощью Boolean(value).

! *Подчиняется правилам*: !

Значение	Становится...
`0`, `null`, `undefined`, `NaN`, `""`	`false`
любое другое значение	`true`

Большую часть из этих правил легко понять и запомнить. Особые случаи, в которых часто допускаются ошибки:

- undefined при численном преобразовании становится NaN, не 0.
- "0" и строки из одних пробелов типа " " при логическом преобразовании всегда true.

1.7 БАЗОВЫЕ ОПЕРАТОРЫ, МАТЕМАТИКА

- *Унарным* называется оператор, который применяется к одному операнду. Например, оператор унарный минус "-" меняет знак числа на противоположный:

```
let x = 1;
x = -x;
alert( x ); // -1, применили унарный минус
```

- *Бинарным* называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

```
let x = 1, y = 3;
alert( y - x ); // 2, бинарный минус вычитает значения
```

Поддерживаются следующие математические операторы:

- Сложение +,
- Вычитание -,
- Умножение *,
- Деление /,
- Взятие остатка от деления %,
- Возведение в степень **.

Первые четыре оператора очевидны, а про % и ** стоит сказать несколько слов.

Взятие остатка %

Оператор взятия остатка %, несмотря на обозначение, никакого отношения к процентам не имеет.

Результат a % b – это остаток от целочисленного деления a на b.

Например:

```
alert( 5 % 2 ); // 1, остаток от деления 5 на 2
alert( 8 % 3 ); // 2, остаток от деления 8 на 3
```

Приведение к числу, унарный +

Плюс + существует в двух формах: бинарной, которую мы использовали выше, и унарной.

Унарный, то есть применённый к одному значению, плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число.

Например:

```
// Не влияет на числа
let x = 1;
alert( +x ); // 1
let y = -2;
alert( +y ); // -2
```

1.8 ОПЕРАТОРЫ СРАВНЕНИЯ

В JavaScript они записываются так:

- Больше/меньше: a > b, a < b.
- Больше/меньше или равно: a >= b, a <= b.
- Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства ==. Один знак равенства a = b означал бы присваивание.

- Не равно. В математике обозначается символом ≠, но в JavaScript записывается как a != b.

Все операторы сравнения возвращают значение логического типа:

- true – означает «да», «верно», «истина».
- false – означает «нет», «неверно», «ложь».

```
alert( 2 > 1 );  // true (верно)
alert( 2 == 1 ); // false (неверно)
alert( 2 != 1 ); // true (верно)
```

! Алгоритм сравнения двух строк довольно прост: **!**

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. Сравнение завершено.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

При сравнении значений разных типов JavaScript приводит каждое из них к числу.

Например:

```
alert( '2' > 1 ); // true, строка '2' становится числом 2
alert( '01' == 1 ); // true, строка '01' становится числом 1
```

Логическое значение true становится 1, а false – 0.

Например:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

Использование обычного сравнения == может вызывать проблемы. Например, оно не отличает 0 от false:

```
alert( 0 == false ); // true
```

Та же проблема с пустой строкой:

```
alert( '' == false ); // true
```

Это происходит из-за того, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём.

Как же тогда отличать 0 от false?

! Оператор строгого равенства === проверяет равенство без приведения типов. !

Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования.

Давайте проверим:

```
alert( 0 === false ); // false, так как сравниваются разные типы
```

Ещё есть оператор строгого неравенства !==, аналогичный !=.

Оператор строгого равенства дольше писать, но он делает код более очевидным и оставляет меньше места для ошибок.

Сравнение с null и undefined

Поведение null и undefined при сравнении с другими значениями — особое:

При строгом равенстве ===

Эти значения различны, так как различны их типы.

```
alert( null === undefined ); // false
```

При нестрогом равенстве ==

Эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

```
alert( null == undefined ); // true
```

При использовании математических операторов и других операторов сравнения < > <= >=

Значения null/undefined преобразуются к числам: null становится 0, а undefined – NaN.

1.9 УСЛОВНОЕ ВЕТВЛЕНИЕ: IF, '?'

Инструкция if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода.

```
if (year == 2015) {
  alert( "Правильно!" );
  alert( "Вы такой умный!" );
}
```

Рекомендовано использовать фигурные скобки {} всегда, когда вы используете инструкцию if, даже если выполняется только одна команда. Это улучшает читабельность кода.

Преобразование к логическому типу

Инструкция if (…) вычисляет выражение в скобках и преобразует результат к логическому типу.

Давайте вспомним правила преобразования типов из главы Преобразование типов:

- Число 0, пустая строка "", null, undefined и NaN становятся false. Из-за этого их называют «ложными» («falsy») значениями.
- Остальные значения становятся true, поэтому их называют «правдивыми» («truthy»).

Таким образом, код при таком условии никогда не выполнится:

```
if (0) { // 0 is falsy
  ...
}
```

…а при таком – выполнится всегда:

```
if (1) { // 1 is truthy
  ...
}
```

Мы также можем передать заранее вычисленное в переменной логическое значение в if, например так:

```
let condition = (year == 2015); // преобразуется к true или false
if (condition) {
```

```
  ...
}
```

Условный оператор „?“

Иногда нам нужно определить переменную в зависимости от условия.

Например:

```
let accessAllowed;
let age = prompt('Сколько вам лет?', '');
if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
alert(accessAllowed);
```

Так называемый «условный» оператор «вопросительный знак» позволяет нам сделать это более коротким и простым способом.

Оператор представлен знаком вопроса ?. Его также называют «тернарный», так как этот оператор, единственный в своём роде, имеет три аргумента.

Синтаксис:

```
let result = условие ? значение1 : значение2;
```

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2.

Например:

```
let accessAllowed = (age > 18) ? true : false;
```

Технически, мы можем опустить круглые скобки вокруг age > 18. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения >.

Этот пример будет делать то же самое, что и предыдущий:

```
// оператор сравнения "age > 18" выполняется первым в любом случае
// (нет необходимости заключать его в скобки)
let accessAllowed = age > 18 ? true : false;
```

Но скобки делают код более читабельным, поэтому мы рекомендуем их использовать.

!*На заметку:* !

В примере выше вы можете избежать использования оператора вопросительного знака ?, т.к. сравнение само по себе уже возвращает true/false:

```
// то же самое
let accessAllowed = age > 18;
```

1.10 ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

В JavaScript есть три логических оператора: || (ИЛИ), && (И) и ! (НЕ).

Несмотря на своё название, данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

1.11 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*.

! *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:

НАЧАЛО	I = 0	ВЫПОЛНЯЕТСЯ ОДИН РАЗ ПРИ ВХОДЕ В ЦИКЛ
условие	i < 3	Проверяется *перед* каждой итерацией цикла. Если оно вычислится в false, цикл остановится.
шаг	i++	Выполняется *после* тела цикла на каждой итерации *перед* проверкой условия.
тело	alert(i)	Выполняется снова и снова, пока условие вычисляется в true.

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*.

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.

! *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, и 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 (...)
```

Вызов break/continue возможен только внутри цикла, и метка должна находиться где-то выше этой директивы.

1.12 КОНСТРУКЦИЯ "SWITCH"

Конструкция switch заменяет собой сразу несколько if.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Конструкция switch имеет один или более блок case и необязательный блок default.

Выглядит она так:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
```

```
    [break]
  case 'value2':  // if (x === 'value2')
    ...
    [break]
  default:
    ...
    [break]
}
```

- Переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее.
- Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).
- Если ни один case не совпал – выполняется (если есть) вариант default.

Пример работы

Пример использования switch (сработавший код выделен):

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Маловато' );
    break;
  case 4:
    alert( 'В точку!' );
    break;
  case 5:
    alert( 'Перебор' );
    break;
  default:
    alert( "Нет таких значений" );
}
```

Здесь оператор switch последовательно сравнит a со всеми вариантами из case.

Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert('В точку!') и далее, до ближайшего break, который прервёт выполнение.

! Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются. !

Пример без break:

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Маловато' );
  case 4:
    alert( 'В точку!' );
```

```
  case 5:
    alert( 'Перебор' );
  default:
    alert( "Нет таких значений" );
}
```

В примере выше последовательно выполнятся три alert:

```
alert( 'В точку!' );
alert( 'Перебор' );
alert( "Нет таких значений" );
```

! **Любое выражение может быть аргументом для switch/case** !

И switch и case допускают любое выражение в качестве аргумента.

Например:

```
let a = "1";
let b = 0;
switch (+a) {
  case b + 1:
    alert("Выполнится, т.к. значением +a будет 1, что в точности равно b+1");
    break;
  default:
    alert("Это не выполнится");
}
```

В этом примере выражение +a вычисляется в 1, что совпадает с выражением b + 1 в case, и следовательно, код в этом блоке будет выполнен.

! *Группировка «case»*

Несколько вариантов case, использующих один код, можно группировать.

Для примера, выполним один и тот же код для case 3 и case 5, сгруппировав их:

```
let a = 2 + 2;

switch (a) {
  case 4:
    alert('Правильно!');
    break;
  case 3: // (*) группируем оба case
  case 5:
    alert('Неправильно!');
    alert("Может вам посетить урок математики?");
    break;
  default:
    alert('Результат выглядит странновато. Честно.');
}
```

Теперь оба варианта 3 и 5 выводят одно сообщение.

Возможность группировать case – это побочный эффект того, как switch/case работает без break. Здесь выполнение case 3 начинается со строки (*) и продолжается в case 5, потому что отсутствует break.

! *Тип имеет значение*

Нужно отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство.

Для примера, давайте рассмотрим следующий код:

```
let arg = prompt("Введите число?");
switch (arg) {
  case '0':
  case '1':
    alert( 'Один или ноль' );
    break;
  case '2':
    alert( 'Два' );
    break;
  case 3:
    alert( 'Никогда не выполнится!' );
    break;
  default:
    alert( 'Неизвестное значение' );
}
```

1. Для '0' и '1' выполнится первый alert.
2. Для '2' – второй alert.
3. Но для 3, результат выполнения prompt будет строка "3", которая не соответствует строгому равенству === с числом 3. Таким образом, мы имеем «мёртвый код» в case 3! Выполнится вариант default.

1.13 ФУНКЦИИ

Объявление функции имеет вид:

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

- Передаваемые значения копируются в параметры функции и становятся локальными переменными.
- Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.
- Функция может возвращать значение. Если этого не происходит, тогда результат равен undefined.

Для того, чтобы сделать код более чистым и понятным, рекомендуется использовать локальные переменные и параметры функций, не пользоваться внешними переменными.

Функция, которая получает параметры, работает с ними и затем возвращает результат, гораздо понятнее функции, вызываемой без параметров, но изменяющей внешние переменные, что чревато побочными эффектами.

Именование функций:

- Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает, и что возвращает.
- Функция – это действие, поэтому её имя обычно является глаголом.
- Есть много общепринятых префиксов, таких как: `create…`, `show…`, `get…`, `check…` и т.д. Пользуйтесь ими как подсказками, поясняющими, что делает функция.

Задание 1

Следующая функция возвращает `true`, если параметр `age` больше `18`.

В ином случае она задаёт вопрос `confirm` и возвращает его результат.

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Родители разрешили?');
  }
}
```

Перепишите функцию, чтобы она делала то же самое, но без `if`, в одну строку.

Сделайте два варианта функции `checkAge`:

1. *Используя оператор `?`*
2. *Используя оператор `||`*

Задание 2

Напишите функцию `min(a,b)`, которая возвращает меньшее из чисел `a` и `b`.

Задание 3

Напишите функцию `pow(x,n)`, которая возвращает `x` в степени `n`.

Задание 4

Напишите функцию, которая принимает первым аргументом число, если число нечетное, то вывести его пользователю (alert), иначе – предлагать пользователю ввести число заново (prompt) до тех пор, пока пользователь не введет нечетное число.

Задание 5

Напишите функцию, которая принимает первым аргументом строку, если строка содержит в себе только цифры, следует умножить число в строке на 2 и вывести на экран, иначе предложить пользователю ввести строку заново до тех пор, пока пользователь не введет число.

Задание 6

Напишите функцию, которая принимает первым аргументом массив и выводит на экран все четные элементы массива.

! Контрольные вопросы !

1. Способы объявления переменных.
2. Отличие let от const.
3. Перечислить вес типы данных в JS
4. На какие две группы можно разделить типы данных?
5. Разница примитивов и объектов?
6. В чем разница == от ===?
7. Способы приведения числа к строке и строки к числу?

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.