Операторы (if, for while и др.)

if

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

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');
if (year < 2015) {
  alert( 'Это слишком рано...' );
} else if (year > 2015) {
  alert( 'Это поздновато' );
} else {
  alert( 'Верно!' );
}
```

Условный оператор „?“

Синтаксис: **let result = условие ? значение1 : значение2;**

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

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

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++;
}
```

Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные скобки {…}.

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.

```
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.

Цикл «for»

```
for (начало; условие; шаг) {
  // ... тело цикла ...
}
```

Например:

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

В примере переменная счётчика i была объявлена прямо в цикле. Вместо объявления новой переменной мы можем использовать уже существующую. Любая часть for может быть пропущена.

Прерывание цикла: «break»

Мы можем выйти из цикла в любой момент с помощью специальной директивы **break**. Например:

```
let sum = 0;
while (true) {
  let value = +prompt("Введите число", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Сумма: ' + sum );
```

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). 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
}
```

Нельзя использовать break/continue справа от оператора „?“

Метки для break/continue

Бывает, нужно выйти одновременно из нескольких уровней цикла сразу. 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 (...) {
  ...
}
```

Например:

```
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++) { ... }
```

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.

Конструкция "switch"

Конструкция switch заменяет собой сразу несколько if

```
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 по всем остальным case (без проверки)**, до ближайшего **break** (или до конца switch). Если ни один case не совпал – выполняется (если есть) вариант default.

Несколько вариантов case, использующих один код, можно группировать. Например:

```
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.');
}
```

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

Еще один способ применения switch:

```
var number = 100;
switch (true) {
    case number > 18 && number < 30:
        console.log("number > 18 и < 30");
        break;
    case number > 50 && number < 70:
        console.log("number > 50 и < 70");
        break;
    default:
        console.log("default value");
}
```

Оператор «запятая»

Оператор «**запятая**» (,) редко применяется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нам нужно знать его, чтобы понимать, что при этом происходит. Оператор «запятая» предоставляет нам возможность вычислять несколько выражений, разделяя их запятой ,. Каждое выражение выполняется, но возвращается **результат только последнего**. Например:

```
let a = (1 + 2, 3 + 4);
alert( a ); // 7 (результат вычисления 3 + 4)
```

Первое выражение 1 + 2 выполняется, а результат отбрасывается. Затем идёт 3 + 4, выражение выполняется и возвращается результат.

! Запятая имеет очень **низкий приоритет**

Пожалуйста, обратите внимание, что оператор , имеет очень низкий приоритет, ниже **=**, поэтому скобки важны в приведённом выше примере. Без них в a = 1 + 2, 3 + 4 сначала выполнится +, суммируя числа в a = 3, 7, затем оператор присваивания = присвоит a = 3, а то, что идёт дальше, будет игнорировано. Всё так же, как в (a = 1 + 2), 3 + 4.

Зачем нам оператор, который отбрасывает всё, кроме последнего выражения? Иногда его используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

```
// три операции в одной строке
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```

Такие трюки используются во многих JavaScript-фреймворках. Вот почему мы упоминаем их. Но обычно они не улучшают читабельность кода, поэтому стоит хорошо подумать, прежде чем их использовать.