Conditional statements[edit]

if

The `if` statement is straightforward — if the given expression is true, the statement or statements will be executed. Otherwise, they are skipped.

```
if (a === b) {
  document.body.innerHTML += "a equals b";
}
```

The `if` statement may also consist of multiple parts, incorporating `else` and `else if` sections. These keywords are part of the `if` statement, and identify the code blocks that are executed, if the preceding condition is false.

```
if (a === b) {
  document.body.innerHTML += "a equals b";
} else if (a === c) {
  document.body.innerHTML += "a equals c";
} else {
  document.body.innerHTML += "a does not equal either b or c";
}
```

while

The `while` statement executes a given statement as long as a given expression is true. For example, the code block below will increase the variable `c` to 10:

```
while (c < 10) {
  c += 1;
  // …
}
```

This control loop also recognizes the `break` and `continue` keywords. The `break` keyword causes the immediate termination of the loop, allowing for the loop to terminate from anywhere within the while block.

The `continue` keyword finishes the current iteration of the `while` block or statement, and checks the condition to see, if it is true. If it is true, the loop commences again.

do … while

The `do … while` statement executes a given statement as long as a given expression is true - however, unlike the `while` statement, this control structure will always execute the statement or block at least once. For example, the code block below will increase the variable `c` to 10:

```
do {
  c += 1;
} while (c < 10);
```

As with `while`, `break` and `continue` are both recognized and operate in the same manner. In other words, `break` exits the loop, and `continue` checks the condition before attempting to restart the loop.

for

The `for` statement allows greater control over the condition of iteration. While it has a conditional statement, it also allows a pre-loop statement, and post-loop increment without affecting the condition. The initial expression is executed once, and the conditional is always checked at the beginning of each loop. At the end of the loop, the increment statement executes before the condition is checked once again. The syntax is:

```
for (<initial expression>;<condition>;<final expression>)
```

The `for` statement is usually used for integer counters:

```
var c;
for (c = 0; c < 10; c += 1) {
  // …
}
```

While the increment statement is normally used to increase a variable by one per loop iteration, it can contain any statement, such as one that decreases the counter.

`break` and `continue` are both recognized. The `continue` statement will still execute the increment statement before the condition is checked.

A second version of this loop is the `for .. in` statement that has following form:

```
for (element in object) {
  // …
}
```

The order of object elements accessed by this version is arbitrary. For instance, this structure can be used to loop through all the properties of an object instance. It should not be used when the object is of Array type

switch

The `switch` statement evaluates an expression, and determines flow control based on the result of the expression:

```
switch(i) {
case 1:
  // …
  break;
case 2:
  // …
  break;
default:
  // …
  break;
}
```

When `i` gets evaluated, its value is checked against each of the `case` labels. These `case` labels appear in the `switch` statement and, if the value for the case matches `i`, continues the execution at that point. If none of the case labels match, execution continues at the `default` label (or skips the switch statement entirely, if none is present.)

Case labels may only have constants as part of their condition.

The `break` keyword exits the `switch` statement, and appears at the end of each case in order to prevent undesired code from executing. While the `break` keyword may be omitted (for example, you want a block of code executed for multiple cases), it may be considered bad practice doing so.

The `continue` keyword does not apply to `switch` statements.

Omitting the `break` can be used to test for more than one value at a time:

```
switch(i) {
case 1:
case 2:
case 3:
  // …
  break;
case 4:
  // …
  break;
default:
  // …
  break;
}
```

In this case the program will run the same code in case `i` equals 1, 2 or 3.

with[edit]

The `with` statement is used to extend the scope chain for a block[1] and has the following syntax:

```
with (expression) {
  // statement
}
```

Pros

The with statement can help to

- reduce file size by reducing the need to repeat a lengthy object reference, and
- relieve the interpreter of parsing repeated object references.

However, in many cases, this can be achieved by using a temporary variable to store a reference to the desired object.

Cons

The with statement forces the specified object to be searched first for all name lookups. Therefore

- all identifiers that aren't members of the specified object will be found more slowly in a 'with' block and should only be used to encompass code blocks that access members of the object.
- `with` makes it difficult for a human or a machine to find out which object was meant by searching the *scope chain*.
- Used with something else than a plain object, `with` may not be forward-compatible.

Therefore, the use of the with statement is not recommended, as it may be the source of confusing bugs and compatibility issues.

Example

```
var area;
var r = 10;

with (Math) {
  a = PI*r*r;        // == a = Math.PI*r*r
  x = r*cos(PI);     // == a = r*Math.cos(Math.PI);
  y = r*sin(PI/2);   // == a = r*Math.sin(Math.PI/2);
}
```