# Chapter 11: Conditions

Conditional expressions, involving keywords such as if and else, provide JavaScript programs with the ability to perform different actions depending on a Boolean condition: true or false. This section covers the use of JavaScript conditionals, Boolean logic, and ternary statements.

## Section 11.1: Ternary operators

Can be used to shorten if/else operations. This comes in handy for returning a value quickly (i.e. in order to assign it to another variable).

For example:

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

In this case, `result` gets the 'cute' value, because the value of animal is 'kitty'. If animal had another value, result would get the 'still nice' value.

Compare this to what the code would like with **`if`**/**`else`** conditions.

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

The `if` or **`else`** conditions may have several operations. In this case the operator returns the result of the last expression.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Because `a` was equal to 0, it becomes 1, and `str` becomes 'not a test'. The operation which involved `str` was the last, so `b` receives the result of the operation, which is the value contained in `str`, i.e. 'not a test'.

Ternary operators *always* expect else conditions, otherwise you'll get a syntax error. As a workaround you could return a zero something similar in the else branch - this doesn't matter if you aren't using the return value but just shortening (or attempting to shorten) the operation.

```
var a = 1;
a === 1 ? alert('Hey, it is 1!') : 0;
```

As you see, `if (a === 1) alert('Hey, it is 1!');` would do the same thing. It would be just a char longer, since it doesn't need an obligatory **`else`** condition. If an **`else`** condition was involved, the ternary method would be much cleaner.

```
a === 1 ? alert('Hey, it is 1!') : alert('Weird, what could it be?');
if (a === 1) alert('Hey, it is 1!') else alert('Weird, what could it be?');
```

Ternaries can be nested to encapsulate additional logic. For example

```
foo ? bar ? 1 : 2 : 3

// To be clear, this is evaluated left to right
// and can be more explicitly expressed as:

foo ? (bar ? 1 : 2) : 3
```

This is the same as the following `if/else`

```
if (foo) {
  if (bar) {
    1
  } else {
    2
  }
} else {
  3
}
```

Stylistically this should only be used with short variable names, as multi-line ternaries can drastically decrease readability.

The only statements which cannot be used in ternaries are control statements. For example, you cannot use return or break with ternaries. The following expression will be invalid.

```
var animal = 'kitty';
for (var i = 0; i < 5; ++i) {
    (animal === 'kitty') ? break:console.log(i);
}
```

For return statements, the following would also be invalid:

```
var animal = 'kitty';
(animal === 'kitty') ? return 'meow' : return 'woof';
```

To do the above properly, you would return the ternary as follows:

```
var animal = 'kitty';
return (animal === 'kitty') ? 'meow' : 'woof';
```

## Section 11.2: Switch statement

Switch statements compare the value of an expression against 1 or more values and executes different sections of code based on that comparison.

```
var value = 1;
switch (value) {
  case 1:
    console.log('I will always run');
    break;
  case 2:
    console.log('I will never run');
    break;
}
```

The `break` statement "breaks" out of the switch statement and ensures no more code within the switch statement is executed. This is how sections are defined and allows the user to make "fall through" cases.

> **Warning**: lack of a `break` or `return` statement for each case means the program will continue to evaluate the next case, even if the case criteria is unmet!

```
switch (value) {
  case 1:
    console.log('I will only run if value === 1');
    // Here, the code "falls through" and will run the code under case 2
  case 2:
    console.log('I will run if value === 1 or value === 2');
    break;
  case 3:
    console.log('I will only run if value === 3');
    break;
}
```

The last case is the `default` case. This one will run if no other matches were made.

```
var animal = 'Lion';
switch (animal) {
  case 'Dog':
    console.log('I will not run since animal !== "Dog"');
    break;
  case 'Cat':
    console.log('I will not run since animal !== "Cat"');
    break;
  default:
    console.log('I will run since animal does not match any other case');
}
```

It should be noted that a case expression can be any kind of expression. This means you can use comparisons, function calls, etc. as case values.

```
function john() {
  return 'John';
}

function jacob() {
  return 'Jacob';
}

switch (name) {
  case john(): // Compare name with the return value of john() (name == "John")
    console.log('I will run if name === "John"');
    break;
  case 'Ja' + 'ne': // Concatenate the strings together then compare (name == "Jane")
    console.log('I will run if name === "Jane"');
    break;
  case john() + ' ' + jacob() + ' Jingleheimer Schmidt':
    console.log('His name is equal to name too!');
    break;
}
```

**Multiple Inclusive Criteria for Cases**

Since cases "fall through" without a `break` or `return` statement, you can use this to create multiple inclusive criteria:

```
var x = "c"
switch (x) {
   case "a":
   case "b":
   case "c":
      console.log("Either a, b, or c was selected.");
      break;
   case "d":
      console.log("Only d was selected.");
      break;
   default:
      console.log("No case was matched.");
      break;  // precautionary break if case order changes
}
```

## Section 11.3: If / Else If / Else Control

In its most simple form, an `if` condition can be used like this:

```
var i = 0;

if (i < 1) {
    console.log("i is smaller than 1");
}
```

The condition `i < 1` is evaluated, and if it evaluates to **`true`** the block that follows is executed. If it evaluates to **`false`**, the block is skipped.

An `if` condition can be expanded with an **`else`** block. The condition is checked *once* as above, and if it evaluates to **`false`** a secondary block will be executed (which would be skipped if the condition were **`true`**). An example:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    console.log("i was not smaller than 1");
}
```

Supposing the **`else`** block contains nothing but another `if` block (with optionally an **`else`** block) like this:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    if (i < 2) {
        console.log("i is smaller than 2");
    } else {
        console.log("none of the previous conditions was true");
    }
}
```

Then there is also a different way to write this which reduces nesting:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else if (i < 2) {
    console.log("i is smaller than 2");
} else {
    console.log("none of the previous conditions was true");
```

```
}
```

Some important footnotes about the above examples:

- If any one condition evaluated to `true`, no other condition in that chain of blocks will be evaluated, and all corresponding blocks (including the `else` block) will not be executed.
- The number of `else if` parts is practically unlimited. The last example above only contains one, but you can have as many as you like.
- The *condition* inside an `if` statement can be anything that can be coerced to a boolean value, see the topic on boolean logic for more details;
- The `if-else-if` ladder exits at the first success. That is, in the example above, if the value of `i` is 0.5 then the first branch is executed. If the conditions overlap, the first criteria occurring in the flow of execution is executed. The other condition, which could also be true is ignored.
- If you have only one statement, the braces around that statement are technically optional, e.g this is fine:

  ```
  if (i < 1) console.log("i is smaller than 1");
  ```

  And this will work as well:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
  ```

  If you want to execute multiple statements inside an `if` block, then the curly braces around them are mandatory. Only using indentation isn't enough. For example, the following code:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
      console.log("this will run REGARDLESS of the condition"); // Warning, see text!
  ```

  is equivalent to:

  ```
  if (i < 1) {
      console.log("i is smaller than 1");
  }
  console.log("this will run REGARDLESS of the condition");
  ```

# Section 11.4: Strategy

A strategy pattern can be used in JavaScript in many cases to replace a switch statement. It is especially helpful when the number of conditions is dynamic or very large. It allows the code for each condition to be independent and separately testable.

Strategy object is simple an object with multiple functions, representing each separate condition. Example:

```
const AnimalSays = {
    dog () {
        return 'woof';
    },

    cat () {
        return 'meow';
    },
```

```
    lion () {
        return 'roar';
    },

    // ... other animals

    default () {
        return 'moo';
    }
};
```

The above object can be used as follows:

```
function makeAnimalSpeak (animal) {
    // Match the animal by type
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' says ' + speak());
}
```

Results:

```
makeAnimalSpeak('dog') // => 'dog says woof'
makeAnimalSpeak('cat') // => 'cat says meow'
makeAnimalSpeak('lion') // => 'lion says roar'
makeAnimalSpeak('snake') // => 'snake says moo'
```

In the last case, our default function handles any missing animals.

## Section 11.5: Using || and && short circuiting

The Boolean operators `||` and `&&` will "short circuit" and not evaluate the second parameter if the first is true or false respectively. This can be used to write short conditionals like:

```
var x = 10

x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")
```