JavaScript 1 - Module 2

Strings and Logic

Arrays

Objects

Functions

Solutions for JS1 Lesson 1.4 Loops

Exercise 1

a) Make a for loop counting from 0 through 10 (including the 10). Console log out the numbers.

b) Make a for loop counting from 6 through 11 (including the 11). Console log out the numbers.

c) Make a for loop counting from 10 through 1 (including the 1). Console log out the numbers.

```
// Exercise 1

// 1a
console.log ("Exercise 1a");
for (let i = 0; i < 11; i++) {
    console.log(i);
}

// 1b
console.log ("Exercise 1b");
for (let i = 6; i <= 11; i++) {
    console.log(i);
}

// 1c
console.log ("Exercise 1c");
for (let i = 10; i > 0; i--) {
    console.log(i);
}
```

Exercise 2

a) Make a while loop counting from 0 through 10 (including the 10). Console log out the numbers.

b) Make a while loop counting from 6 through 11 (including the 11). Console log out the numbers.

c) Make a while loop counting from 10 through 1 (including the 1). Console log out the numbers.

```
// Exercise 2

// 2a
console.log ("Exercise 2a");
let i = 0;
while (i <= 10) {
    console.log(i);
    i++;
}

// 2b
console.log ("Exercise 2b");
i = 6;
while (i <= 11) {
    console.log(i);
    i++;
}

// 2c
console.log ("Exercise 2c");
i = 10;
while (i > 0) {
    console.log(i);
    i--;
}
```

Note:

`const` declarations cannot be redeclared by any other declaration in the same scope. (And cannot be reassigned.)

`let` declarations cannot be redeclared by any other declaration in the same scope. (But may be reassigned.)

Duplicate variable declarations using `var` will not trigger an error, even in strict mode, and the variable will not lose its value, unless the declaration has an initializer:

```
var a = 1;
var a = 2;
console.log(a); // 2
var a;
console.log(a); // 2; not undefined
```

We'll talk more about Scope and Hoisting (for functions) later.

Exercise 3

a) Make a for loop counting from 1 through 10. Console log out the numbers, except for 5 (let the loop skip over 5).

Tip look at continue

b) Make a while loop counting down from 10 through 0, but stops after 2 is reached. Console log out the numbers (10-2).

> Tip look at break

```
// Exercise 3

// 3a
console.log ("Exercise 3a");
for (let i = 1; i <= 10; i++) {
    if (i === 5) { continue; }
    console.log(i);
}

// 3b
console.log ("Exercise 3b");
for (let i = 10; i >= 0; i--) {
    if (i < 2) { break; }
    console.log(i);
}

// 3b now with a while loop :-p
console.log ("Exercise 3b, 2nd try");
i = 10;
while (i >= 0) {
    console.log(i);
    if (i === 2) { break; } // Break after logging the 2.
    i--;
}
```

Exercise 4

a) Make a do...while loop counting down from 10 through 5. Console log out the numbers.

b) What becomes the output if you set the counter to 0 before the loop?

```
// Exercise 4

// 4a
console.log ("Exercise 4a");
i = 10;
do {
    console.log(i);
    i--;
} while (i >= 5);

// 4b
console.log ("Exercise 4b");
i = 0;
do {
    console.log(i);
    i--;
} while (i >= 5);
```

Exercise 5

Make a for loop that counts just the even numbers (0, 2, 4, 6, ...) from 0 through 20.

Console log out the numbers.

Tip adjust the iteration in the final expression.

```
// Exercise 5

console.log ("Exercise 5");
for (let i = 0; i <= 20; i+=2) {
    console.log(i);
}

// Alternative:
for (let i = 0; i <= 20; i++) {
    if (i % 2 == 0) {
        console.log(i);
    }
}
```

Exercise 6

Given an array:

```
const catBreeds = ["Abyssinian", "Balinese", "Birman", "Chartreux", "Egyptian Mau", "Maine Coon", "Norwegian Forest Cat", "Ragdoll", "Siamese", "Siberian"];
```

Use a `for...of` loop and console log out the diffent cats with the same lead text for each breed.

```
// Exercise 6

console.log ("Exercise 6");
const catBreeds = ["Abyssinian", "Balinese", "Birman",
"Chartreux", "Egyptian Mau", "Maine Coon",
"Norwegian Forest Cat", "Ragdoll", "Siamese", "Siberian"];

for (let cat of catBreeds) {
    console.log(cat + " is a cat breed.");
}
```

Exercise 7 - Level 2

Using a double loop, make the following pattern:

```
*
**
***
****
*****
```

```
// Exercise 7 - Level 2
console.log ("Exercise 7 - Level 2");
let text = "";
const rows = 5;
// Outer for = rows
for (let i = 1; i <= rows; i++) {
    // Inner for = columns
    for (let j = 1; j <= i; j++) {
        text += "*";
    }
    text += "\n"; // Add a line break for each row
}
console.log (text);

// What happens if you change the inner loop to
// for (let j = rows; j >= i; j--)
```

Excercize 8 - Level 2

Fizz buzz is a group word game for children to teach them about division. Players take turns to count incrementally, replacing any number divisible by three with the word "fizz", and any number divisible by five with the word "buzz".

Players generally sit in a circle. The player designated to go first says the number "1", and the players then count upwards in turn. However, any number divisible by three is replaced by the word fizz and any number divisible by five by the word buzz. Numbers divisible by 15 become fizz buzz.

Excercize 8 - Level 2, cont.

a) Using a for loop, make a variant of Fizz buzz where you count from 1 through 20 and add each number to a text string as you count. In the end console log out the answer string.

The final answer should look like this: "1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz Buzz, 16, 17, Fizz, 19, Buzz"

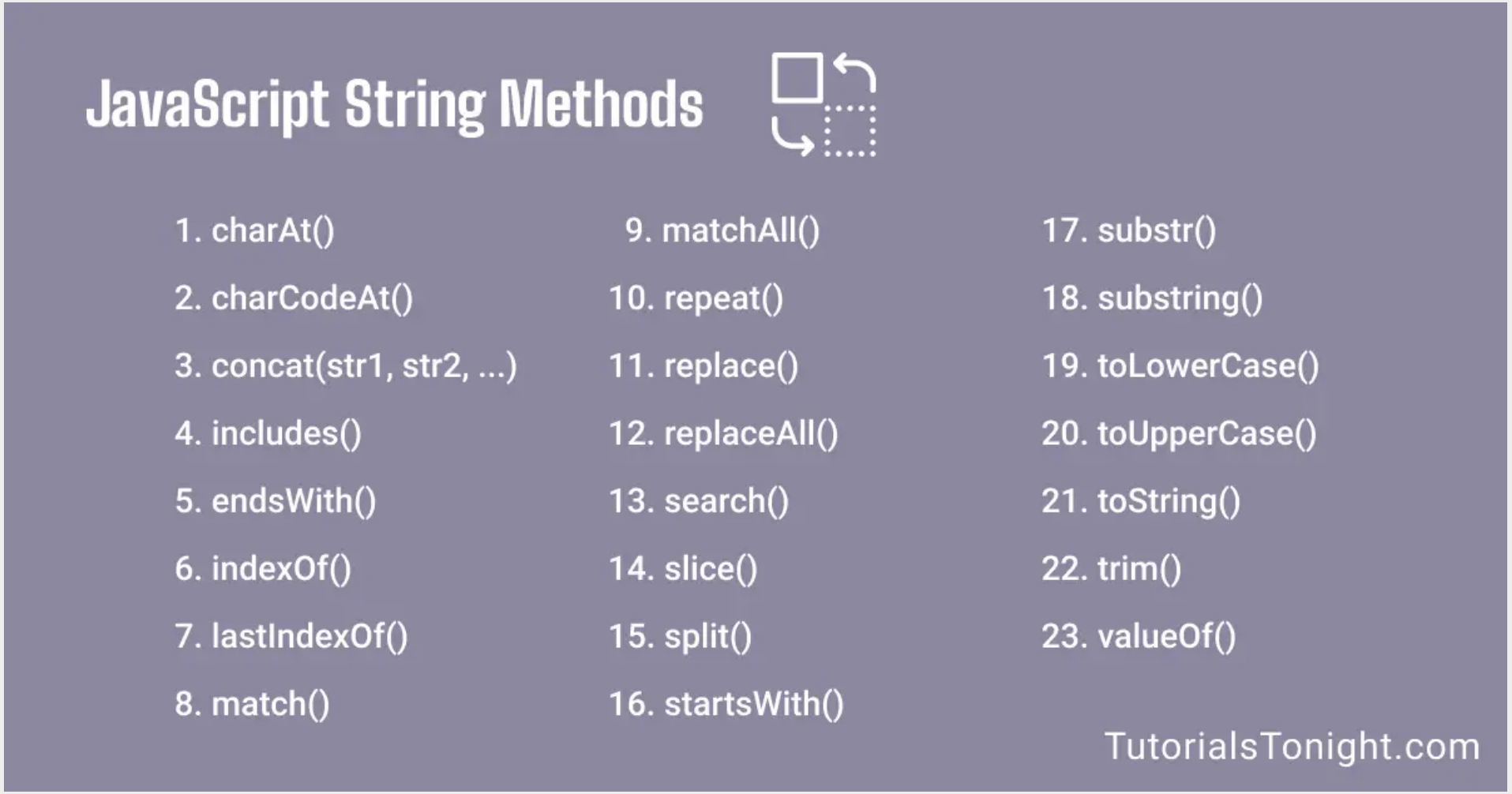
b) Make a variant where you count from 1 through 30, and buzz is used for numbers divisible by 7 (not 5).

```
// Exercise 8 - Level 2
console.log ("Exercise 7 - Level 2");
const max = 20; // a = 20, b = 30
let output = "";
for (let i = 1; i <= max; i++) {

    if (i%3 === 0 || i%5 === 0) { // for b, swap 5 for 7
        if (i%3 === 0) {
            output += "Fizz";
        }
        if (i%5 === 0) { // for b, swap 5 for 7
            output += "Buzz";
        }
    } else {
        output += i;  // If neither fizz or buzz, output the number
    }

    if (i < max) {
        output += ", "; // Add comma for all but the last
    }
}
console.log (output);
```

> Note that `(i%3 === 0)` and `(i%5 === 0)` reapeats, they could be made into variables:

```
console.log ("Exercise 7 - Level 2");
let fizz, buzz;
const maxB = 30; // a = 20, b = 30
let outputB = "";
for (let i = 1; i <= maxB; i++) {
    fizz = (i%3 === 0); // fizz is true if i%3 == 0 (i is divisible into 3)
    buzz = (i%7 === 0); // for b, swap 5 for 7, just here.
    if (fizz || buzz) {
        if (fizz) {
            outputB += "Fizz";
        }
        if (fizz && buzz) {
            outputB += " ";
            // Space between fizz and buzz when both are written
        }
        if (buzz) {
            outputB += "Buzz";
        }
    } else {
        outputB += i; // If neither fizz or buzz, output the number
    }

    if (i < maxB) {
        outputB += ", "; // Add comma for all but the last
    }
}
console.log (outputB);
```

String properties and methods



Source

JavaScript Strings

JavaScript strings are used for storing and manipulating text.

```
let txt = "Whatever you want";
```

JavaScript String Methods

String methods help you to work with strings.

Primitive values, like "Whatever you want", normally cannot have properties or methods (because they are not objects).

But with **JavaScript**, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

String Length

To find the length of a string, use the built-in `length` property:

```
let greeting = "Hello, world!";
console.log(greeting.length); // 13
console.log("greeting".length); // 8
```

Another example:

```
const str = 'Life, the universe and everything. Answer:';

console.log(`${str} ${str.length}`);
// expected output:
// "Life, the universe and everything. Answer: 42"
```

> We will look more at **Template literals** aka **Template strings** later in this lesson.

Finding a String in a String

The `indexOf()` method returns the index of (the position of) the first occurrence of a specified text in a string:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate");
console.log(pos); // 7
```

> JavaScript counts positions from zero. 0 is the first position in a string, 1 is the second, 2 is the third ...

Finding a String in a String, cont.

The `lastIndexOf()` method returns the index of the last occurrence of a specified text in a string:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("locate");
console.log(pos); // 21
```

Both `indexOf()`, and `lastIndexOf()` return -1 if the text is not found.

```
var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("Blåbærsyltetøy");
console.log(pos); // -1
```

Finding a String in a String, cont.

Both `indexOf()`, and `lastIndexOf()` accept a second parameter as the starting position for the search:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate", 15);
console.log(pos); // 21
```

The `lastIndexOf()` methods searches backwards (from the end to the beginning), meaning: if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string.

```
var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("locate", 15);
console.log(pos); // 7
```

Searching for a String in a String

The `search()` method searches a string for a specified value and returns the position of the match:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.search("locate");
```

Does this mean the two methods, `indexOf()` and `search()`, are equal? They accept the same arguments (parameters), and return the same value?

The two methods are NOT equal. These are the differences:

- The `search()` method cannot take a second start position argument.
- The `indexOf()` method cannot take powerful search values (regular expressions).

Extracting String Parts: The `slice()` Method

`slice()` extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: the start position, and the end position (end not included).

This example slices out a portion of a string from position 7 to position 12 (13 minus 1):

```
var str = "Apple, Banana, Kiwi";
var res = str.slice(7, 13);
console.log(res); // "Banana"
```

> Remember: JavaScript counts positions from zero. First position is 0.

Extracting String Parts: The `slice()` Method, cont.

If a parameter is negative, the position is counted from the end of the string.

```
var str = "Apple, Banana, Kiwi";
var res = str.slice(-12, -6);
console.log(res); // "Banana"
```

If you omit the second parameter, the method will slice out the rest of the string:

```
var res = str.slice(7);
console.log(res); // "Banana, Kiwi"

var res = str.slice(-12);
console.log(res); // "Banana, Kiwi"
```

Extracting String Parts: The `substring()` Method

`substring()` is similar to `slice()`.

The difference is that `substring()` cannot accept negative indexes.

```
var str = "Apple, Banana, Kiwi";
var res = str.substring(7, 13);
console.log(res); // "Banana"
```

If you omit the second parameter, `substring()` will slice out the rest of the string.

Extracting String Parts: The `substr()` Method, cont.

The difference from `slice()` is that the second parameter specifies the length of the extracted part.

```
var str = "Apple, Banana, Kiwi";
var res = str.substr(7, 6);
console.log(res); // "Banana"
```

If you omit the second parameter, substr() will slice out the rest of the string.

```
var str = "Apple, Banana, Kiwi";
var res = str.substr(7);
console.log(res); // "Banana, Kiwi"
```

If the first parameter is negative, the position counts from the end of the string.

```
var str = "Apple, Banana, Kiwi";
var res = str.substr(-4);
console.log(res); // "Kiwi"
```

Replacing String Content

The `replace()` method replaces a specified value with another value in a string:

```
var str = "Windows is the best operating system";
var newStr = str.replace("Windows", "macOS");
console.log(newStr); // "macOS is the best operating system"
```

> The `replace()` method does not change the string it is called on. It returns a new string.

By default, the replace() method replaces only the first match:

```
var str = "New York, New York, by Grandmaster Flash and the Furious Five";
var newStr = str.replace("New York", "The Message");
console.log(newStr); // "The Message, New York, by Grandmaster Flash and the Furious Five"
```

> By default, the replace() method is case sensitive. To replace case insensitive, use a *regular expression*.

Converting to Upper and Lower Case

A string is converted to upper case with `toUpperCase()`:

```
var text1 = "Hello World!";
var text2 = text1.toUpperCase();
console.log(text2); // "HELLO WORLD!"
```

A string is converted to lower case with `toLowerCase()`:

```
var text1 = "Hello World!";
var text2 = text1.toLowerCase();
console.log(text2); // "hello world!"
```

The concat() Method

`concat()` joins two or more strings:

```
var text1 = "Hello";
var text2 = "World";
var text3 = text1.concat(" ", text2);
console.log(text3); // "Hello World"
```

The `concat()` method can be used instead of the plus operator. These two lines do the same:

```
var text = "Hello" + " " + "World!";
var text = "Hello".concat(" ", "World!");
```

> All string methods return a new string. They don't modify the original string.

String.trim()

The `trim()` method removes whitespace from both sides of a string:

```
var str = "       Hello World!        ";
var trimmed = str.trim();
console.log(trimmed); // "Hello World!"
```

Extracting String Characters: The `charAt()` Method

The charAt() method returns the character at a specified index (position) in a string:

```
var str = "HELLO WORLD";
var myChar = str.charAt(0); // returns 'H' to myChar
console.log(myChar); // "H"
```

Extracting String Characters: The `charCodeAt()` Method

The `charCodeAt()` method returns the **unicode** of the character at a specified index in a string:

```
var str = "HELLO WORLD";
var myCode = str.charCodeAt(0); // returns 72, unicode for 'H'
console.log(myCode); // 72
```

> The method returns a UTF-16 code
>
> (an integer between 0 and 65535).

Extracting String Characters: Property Access

ECMAScript 5 (2009) allows property access `[ ]` on strings

```
var str = "HELLO WORLD";
console.log(str[0]); // returns 'H'
console.log(str[1]); // returns 'E'
console.log(str[2]); // returns 'L'
console.log(str[3]); // returns 'L'
console.log(str[4]); // returns 'O'
console.log(str[5]); // returns ' '
console.log(str[6]); // returns 'W'
// etc.
```

Property access might be a little unpredictable:

- It makes strings look like arrays (but they are not)
- If no character is found, [ ] returns undefined, while charAt() returns an empty string.
- It is read only. str[0] = "A" gives no error (but does not work!)

```
var str = "HELLO WORLD";
str[0] = "A"; // Gives no error, but does not work
console.log(str[0]); // returns 'H'
```

If you want to work with a string as an array, you can convert it to an array, using the `split()` method.

Template literals (Template strings)

Template literals are string literals allowing *embedded expressions*. You can use *multi-line strings* and *string interpolation* features with them.

Used to be called *Template strings*, but renamed **Template literals** in ECMA2015.

Normal string:

```
var s = "This is i string and a " + variable;
```

Template literal:

```
var t = `This is a string with a ${variable}`;
```

Template literals, cont.

Template literals are enclosed by the backtick (` `) (grave accent) character instead of double or single quotes.

Template literals can contain placeholders. These are indicated by the dollar sign and curly braces, `${expression}`.

The expressions in the placeholders and the text between the backticks (` `) get passed to a function. The default function just concatenates the parts into a single string.

Multi-line strings

Any newline characters inserted in the source are part of the template literal. Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
console.log('string text line 1\n' +
'string text line 2');
// "string text line 1
// string text line 2"
```

Using template literals, you can do the same like this:

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

Expression interpolation

In order to embed expressions within normal strings, you would use the following syntax:

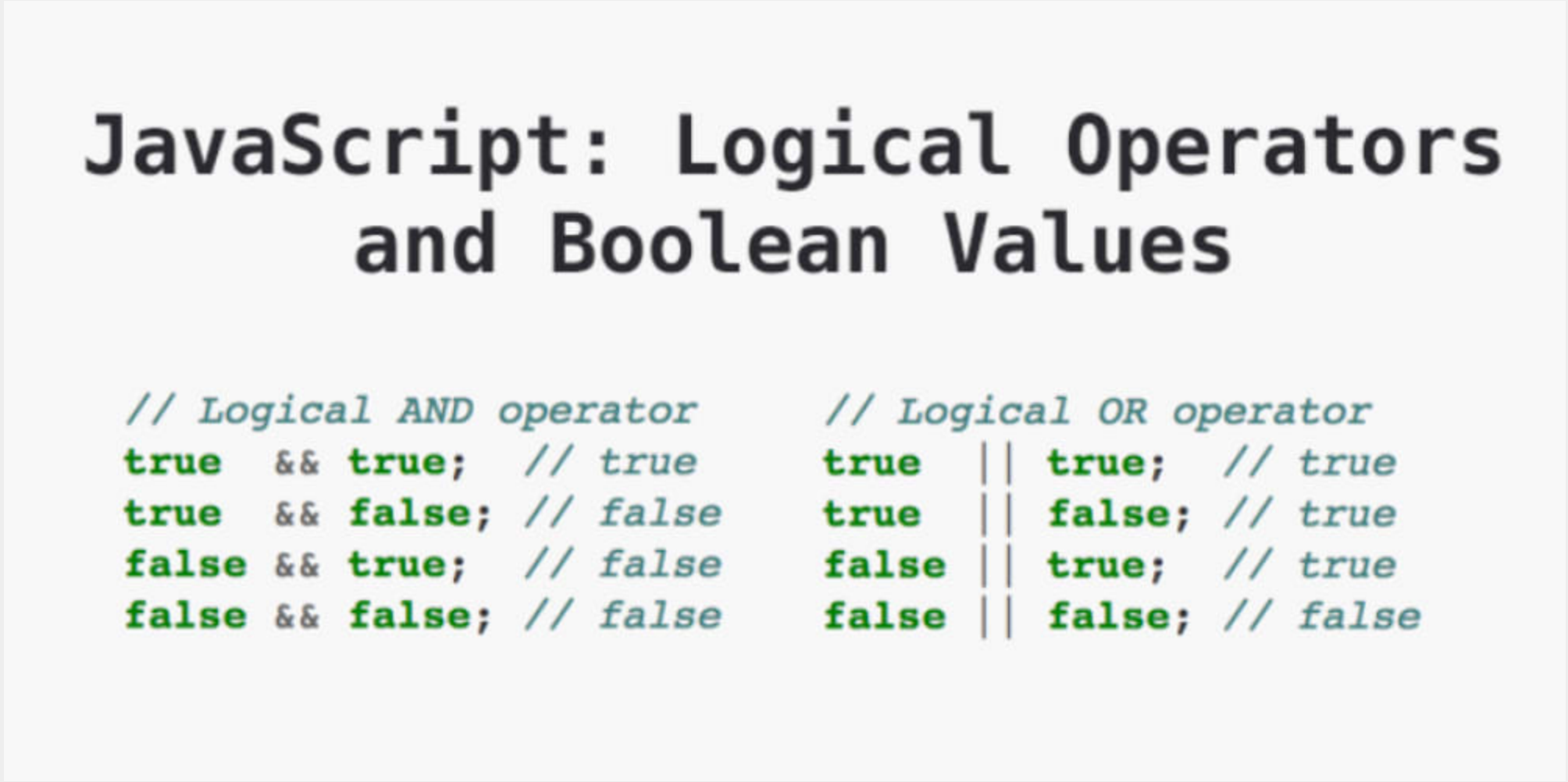
```
let a = 5;
let b = 10;
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
// "Fifteen is 15 and
// not 20."
```

Now, with template literals, you are able to make use of the placeholders, making substitutions like this more readable:

```
let a = 5;
let b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

Break?

Logical Operators



Source

Comparison Operators (R)

Comparison operators are used in logical statements to determine equality or difference between variables or values.

```
var x = 5;
x == 8    // false
x == 5    // true
x == "5"  // true
x === 5   // true
x === "5" // false
x != 8    // true
x !== 5   // false
x !== "5" // true
x !== 8   // true
x > 8     // false
x < 8     // true
x >= 8    // false
x <= 8    // true
```

Logical Operators (R)

Logical operators are used to test for `true` or `false`, to determine the logic between variables or values:

```
var x = 6, y = 3;

// AND &&
(x < 10 && y > 1) // true

// OR ||
(x == 5 || y == 5) // false

// NOT !
!(x == y) // true
```

Logical AND (`&&`)

`expr1 && expr2` returns `expr1` if it can be converted to `false`; otherwise, returns `expr2`.

Thus, when used with Boolean values, **`&&` returns true if both operands are true**; otherwise, returns false.

```
var a1 =  true && true;     // t && t returns true
var a2 =  true && false;    // t && f returns false
var a3 = false && true;     // f && t returns false
var a4 = false && (3 == 4); // f && f returns false
var a5 = 'Cat' && 'Dog';    // t && t returns Dog
var a6 = false && 'Cat';    // f && t returns false
var a7 = 'Cat' && false;    // t && f returns false
```

Logical OR (`||`)

`expr1 || expr2` returns `expr1` if it can be converted to `true`; otherwise, returns `expr2`.

Thus, when used with Boolean values, `||` **returns true if either operand is true**; if both are false, returns false.

```
var o1 =  true || true;     // t || t returns true
var o2 = false || true;     // f || t returns true
var o3 =  true || false;    // t || f returns true
var o4 = false || (3 == 4); // f || f returns false
var o5 = 'Cat' || 'Dog';    // t || t returns Cat
var o6 = false || 'Cat';    // f || t returns Cat
var o7 = 'Cat' || false;    // t || f returns Cat
```

Logical NOT (`!`)

`!expr` returns `false` if its single operand that can be converted to `true`; otherwise, returns `true`.

```
var n1 = !true;  // !t returns false
var n2 = !false; // !f returns true
var n3 = !'Cat'; // !t returns false
```

Short-circuit evaluation

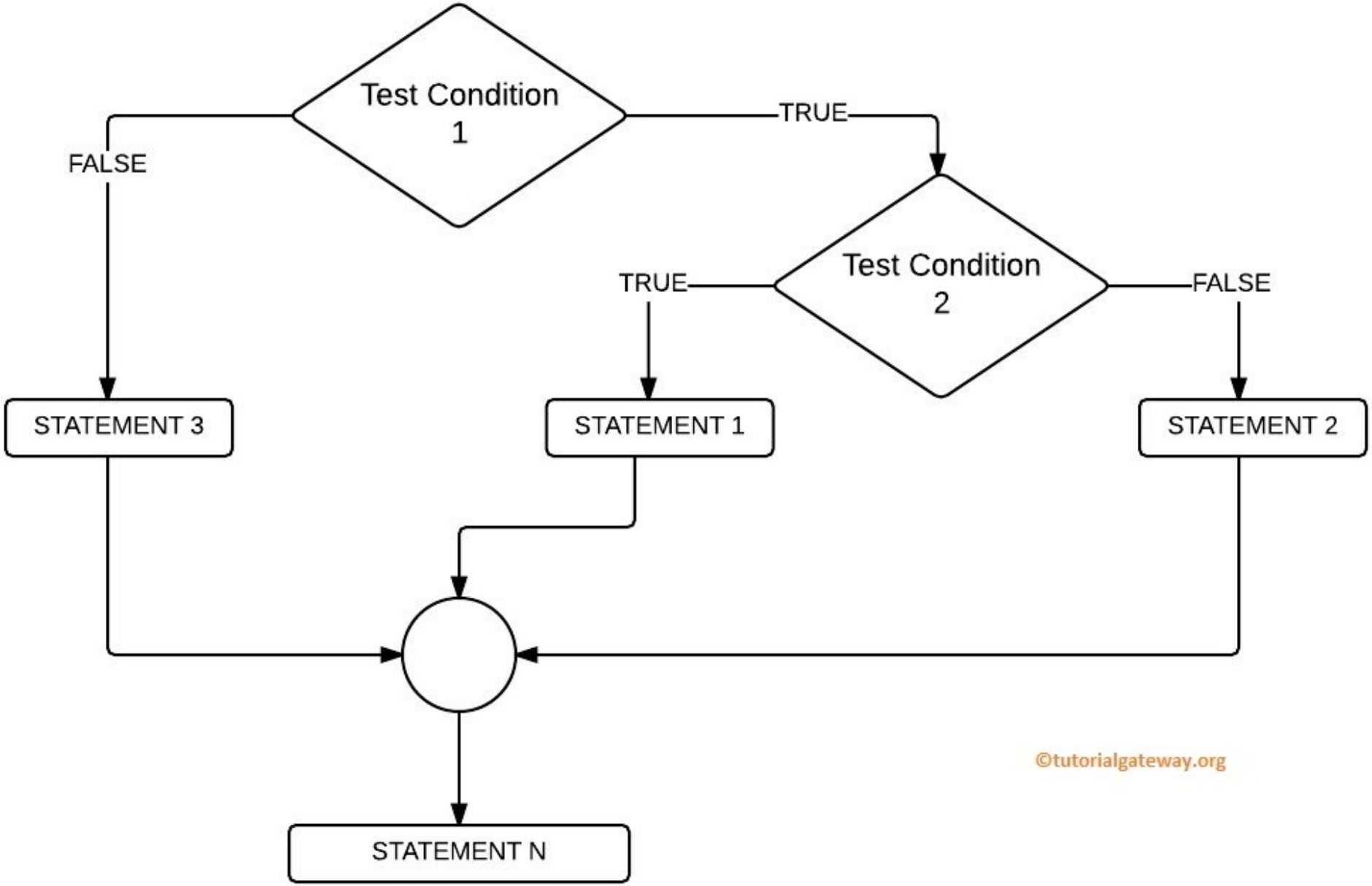
As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false && anything` is short-circuit evaluated to **`false`**.
- `true || anything` is short-circuit evaluated to **`true`**.

The rules of logic guarantee that these evaluations are always correct. Note that the anything part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

Nested if...else (R)

```
if ( Test condition 1 ) {

    //If the Test condition 1 is TRUE then it will check for test condition 2:
    if ( Test condition 2 ) {

        // If the Test condition 2 is TRUE then these statements will be executed:
        STATEMENT 1;

    } else {

        // If the Test condition 2 is FALSE then these statements will be executed:
        STATEMENT 2;

    }

} else {

    // If the test condition 1 is FALSE then these statements will be executed:
    STATEMENT 3;

}

//
STATEMENT N;
```


Test Condition 1
TRUE
FALSE
Test Condition 2
TRUE
FALSE
STATEMENT 3
STATEMENT 1
STATEMENT 2
STATEMENT N
©tutorialgateway.org

Source

Nested if, another example:

```
var x = 14;
var t;

if (x > 10) {
    t = x + " is above 10, ";
    if (x > 20) {
        t += "and also above 20!";
    } else {
        t += "but not above 20.";
    }
} else {
    t = x + " is below 10.";
}
console.log (t);
```

> Try it with different values for x

Sources and Resources

JavaScript Strings

JavaScript String Methods

JS String Reference

JavaScript reference > Standard built-in objects > String

Template literals (Template strings)

Logical operators

Todos

Github Classroom

JS1 Lesson 2.1 Strings and Logic

Mollify

Read String properties and methods, multiple if conditions and nested if statements and do the Lesson Task.