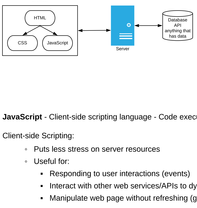
06_JavaScript



JavaScript - Client-side scripting language - Code executes in browser

Client-side Scripting:

- Puts less stress on server resources
- Useful for:
  - Responding to user interactions (events)
  - Interact with other web services/APIs to dynamically update pages
  - Manipulate web page without refreshing (getting a new copy from server)

aka **ECMAScript 6** or **ES6** *(European Computer Manufacturers Association)* - defined standard for modern JavaScript

JavaScript may be embedded in an HTML document using the `<script>` tag or in an external file. *(common/preferred/best practice)*.

Similar syntax to C# and Java (the only thing in common with Java is 1st four letters of name)

Main differences from C# and Java:

- Java/C# require a **runtime environment** - JavaScript requires a **browser**
- Java/C# is **compiled** - JavaScript is **interpreted**
- Java/C# is **statically typed** - JavaScript is **dynamically typed**

Statically typed *(strongly typed)* - data type of variable is declared before use and cannot be changed

Dynamically typed *(loosely typed)* - data type of variable need not be declared. Data type is determined when used based on current content/value stored in the variable.

Naming rules for JavaScript variables

- Variable names are comprised of letters `A-Z`, `a-z`, `_`, `$`, and digits `0-9`.
- Variable names must start with `a letter`, `_`, or `$`.
- Variable names are case-sensitive.
- Variable names may be not be a reserved keyword.

The following are considered best practice when define variables in JavaScript:

- Use *camelCase* for multi-word variable names.
- Use UPPERCASE for constants and separate words with an underscore `_`
- Boolean variable should begin with `is`
- Use `let` to define a variable (same scope as in Java and C#)
- Use `const` to define a constant
- Avoid use of `var` (scope is more complicated than in Java and C#)

Additional, basic, facts concerning JavaScript:

JavaScript statements may or may not end with a `;`

`{}` are used to enclose a self-contained block of code

Numeric literals/values are coded as you would as a human:

Whole number ---> `10` `100` `42` `-23`

Decimal number ---> `1.23` `3.14` `-12.23`

Words/Characters (string) ---> `'Hi there'` `"Hi There"` `"Frank's Class"`

Boolean value ---> `true` `false`

Intentional lack of value/unknown value ---> `Null`

Unintentional lack of value ---> `undefined`

Result is not numeric and should be --> **NaN (Not a Number)**

Extrememly large/small value ---> `infinity` (usually caused by mathematical error such as divide by 0)

Basic `if` statement - ask a question/make a decision

Whenever a question needs to be asked or a decison made by a program, the basic `if` statement is a common solution. There are other conditional statements you wil learn about later.

Basic `if` statement syntaxes:

Perform different processing depending on the condition being true or false:

```
if (condition) {
    processing-when-condition-is-true
}
then {
    processing-when-condition-is-false
}
```

Perform processing only if the condition true:

```
if (condition) {
    processing-when-condition-is-true
}
```

`if` statement may be nested to any number of levels:

```
if (condition) {
        processing-when-condition-is-true
    }
    then {
        if (condition) {
            processing-when-condition-is-true
        }
        then {
            processing-when-condition-is-false
            }
    }

  if (condition) {
            if (condition) {
                processing-when-condition-is-true
            }
            then {
                processing-when-condition-is-false
            }

    }
    then {
        if (condition) {
            processing-when-condition-is-true
        }
        then {
            processing-when-condition-is-false
            }
    }
```

Nested `if` statements can get confusing and hard to understand quickly. Use sparingly and with caution. Other condition statements exist to make nested condition processing easier.

Always be sure you have coded your code block `{}` correctly and you do not have a misplaced `{` or `}`

`==` vs `===`
(is equal) *(strictly equal)*

`==` - compares values and ignores data types
`===` - compares values and data types

Functions in JavaScript

A function is a self-contained unit of code used to perform common processing or to separate a program into logical processing units.

Functions start with the word `function` followed by the function name and optional parameters.

A *parameter* is a data value to be used in the processing of the function.

The processing to be done in athe function is enclosed in `{}`

They don't have a *return type* and the naming convention is *camelCase*.

Functions represent the value return by the function.

Functions generally return a value which replaces the place in the code the function was called.

The `return` statement terminates a function with an optional return value.

The function will also terminate when the ending `}` for the function is encountered.

If a function does not return a value the function value is `undefined`.

When a function terminates, the value returned by the function replaces the function call.

A function may be called anywhere a variable may be coded.

Example of a function to receive two parameters and return their sum:

```
function addem(num1, num2) {

  return num1 + num2

}
```

Example of a calling the function defined above:
`addem(2,3)` `-->` this will be replaced by the value 5

In general, a word followed by a `(` is a function name if not `if`, `for`, `while` or `switch`

Arrays in JavaScript:

Arrays are a series of variables accessable via their relative location (index) in the series.

Variables in an *array* are referred to as **elements**.

In JavaScript an array is defined using `[]` with optional initial values coded inside th `[]`

```
let charles = [10, 20, 30]  // an array of 3 elements
```

To reference the elements in an array: `arrayName[index]`

index values start at 0 (ie. the first element is at index **0**, second element at index **1**)

```
charles[1] --> 20

charles[0] --> 10

charles[2] --> 30

charles[3] --> error!  Index value out of range
```

It is the programmers **responsibility to ensure any index value used is within the bounds of the array!**

`arrayName.length` will return the size *(number of elements)* of the array

The largest allowable index for an array may be computed: `arrayName.length - 1`

`for` statement - loop through a process a specific number of times

Use a `for`-loop to process an array from the beginning to the end

A `for`-loop has 3-parts: `for (initialization; condition; increment)` and a body enclosed in `{}` which is the processing to be done to an element in the array using the loop-index as an index to access the current element in the array.

The loop-index is the variable **initialized**, **tested in the condition** and **incremented**.

initialization - done once at the start of the process

condition - is checked before each loop - controls how many times the loop is executed

increment - done at the end of loop body (just before it goes back & checks condition)

A `for`-loop will execute the statements in the loop body as long as the *condition* is true

When processing an array from beginning to end:

initialization - set loop-index to 0

condition - loop as long as the index is inside array (`loop-index < arrayName.length`)

increment - add 1 to the loop index-index

Example: *(pretty much every for-loop to process all elements will look like this - different array)*

```
for (let i=0; i < arrayName.length; i++) {

  // do something with the arrayName[i] - process the current element

}
```

`let i=0` - define and set the loop-index to 0 - start at the first element in the array

`i < arrayName.length` - keep the index inside the array (max value for i is length-1)

`i++` - increment i (add 1 to loop index) --> `i = i+1` or `i+=1` ok too

Naming the loop-index `i` is a tradition, you can name it anything you want.