JavaScript - Dynamic client-side scripting

JavaScript first steps

JavaScript is a programming language that allows you to implement complex functionalities on web pages. Anytime a webpage is not just static, JavaScript is used.

JavaScript is included in a page by a script tag that either uses the text of the tag as code or loads it from a file referred to with an HTML attribute. The code at the top level (outside any function) is only able to manipulate the elements that are created before the code is loaded.

What is JavaScript?

Button that asks for a new name when pressed:

```
<button type="button">Player 1: Chris</button>
```

```
const button = document.querySelector("button");

button.addEventListener("click", updateName);

function updateName() {
  const name = prompt("Enter a new name");
  button.textContent = `Player 1: ${name}`;
}
```

What can I do with JavaScript?

- Store useful values inside variables. In the above example for instance, we ask for a new name to be entered then store that name in a variable called name.
- Operations on pieces of text (known as "strings" in programming). In the above example we take the string "Player 1: " and join it to the name variable to create the complete text label, e.g. "Player 1: Chris".
- Running code in response to certain events occurring on a web page. We used a click event in our example above to detect when the label is clicked and then run the code that updates the text label.
- ...

Application Programming Interfaces (APIs): functionallity built on top of the client-side JavaScript language.

More advanced metadata can be specified such as the details of a link to be shown when the page is shared on Facebook for example.

- Browser APIs are built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things.

- Third party APIs are not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web.

A very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface, via the Document Object Model API (your code (HTML, CSS, Javascript is executed inside an execution environment (the browser tab))). Note that the code in your web documents is generally loaded and executed in the order it appears on the page. Errors may occur if JavaScript is loaded and run before the HTML and CSS that it is intended to modify. Each browser tab has its own separate bucket for running code in (these buckets are called "execution environments" in technical terms) — this means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab — or on another website. This is a good security measure.

Interpreted vs compiled code:

Javascript running order: top to bottom: interpreted language. **Just-in-time compiling** to improve performance; the JavaScript source code gets compiled into a faster, binary format while the script is being used, so that it can be run as quickly as possible. However, JavaScript is still an interpreted language.

Client side vs Server side:

client-side JavaScript vs JavaScript can also be used as a server-side language, for example in the popular Node.js environment

Dynamic vs static code:

The word dynamic is used to describe both client-side JavaScript, and server-side languages — it refers to the ability to update the display of a web page/app to show different things in different circumstances, generating new content as required. A web page with no dynamically updating content is referred to as static — it just shows the same content all the time.

How to add JavaScript to your page?

```
<script>
    // JavaScript goes here
  </script>
```

Example: when you click the button, a new paragraph is generated and placed below, containing text "You clicked the button!".

```
<script>
    document.addEventListener("DOMContentLoaded", () => {
  function createParagraph() {
    const para = document.createElement("p");
    para.textContent = "You clicked the button!";
    document.body.appendChild(para);
  }
```

```
  const buttons = document.querySelectorAll("button");

  for (const button of buttons) {
    button.addEventListener("click", createParagraph);
  }
});
</script>
```

OR put the JavaScript in a separate file and link it to the HTML file

```
<script src="script.js" defer></script>
```

inside script.js add:

```
function createParagraph() {
    const para = document.createElement("p");
    para.textContent = "You clicked the button!";
    document.body.appendChild(para);
  }

  const buttons = document.querySelectorAll("button");

  for (const button of buttons) {
    button.addEventListener("click", createParagraph);
  }
```

Selecting all elements of a group on a page: querySelectorAll():

```
const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

Script loading strategies

In what we have seen before the JavaScript is loaded and run in the head of the document, before HTML body is parsed.

- In the internal example: Now we add an event listener, making sure that the HTML body is completely loaded and parsed, before running the JavaScript inside this block.

```
document.addEventListener("DOMContentLoaded", () => {
    // …
```

```
});
```

- In the external example: add 'defer' attribute, which tells the browser to continue downloading the HTML content once the `script` tag element has been reached.

```
<script src="script.js" defer></script>
```

- An old-fashioned solution to this problem used to be to put your script element right at the bottom of the body (e.g. just before the tag), so that it would load after all the HTML has been parsed. The problem with this solution is that loading/parsing of the script is completely blocked until the HTML DOM has been loaded. On larger sites with lots of JavaScript, this can cause a major performance issue, slowing down your site.

async and defer: Scripts loaded using the async attribute will download the script without blocking the page while the script is being fetched. However, once the download is complete, the script will execute, which blocks the page from rendering. This means that the rest of the content on the web page is prevented from being processed and displayed to the user until the script finishes executing. You get no guarantee that scripts will run in any specific order. It is best to use async when the scripts in the page run independently from each other and depend on no other script on the page.

Scripts loaded with the defer attribute will load in the order they appear on the page. They won't run until the page content has all loaded, which is useful if your scripts depend on the DOM being in place (e.g. they modify one or more elements on the page).

![Alt text](C:\Users\vrank\OneDrive - Vlerick Business School\Nova SBE\Web and Cloud Computing\Summary HTML and JavaScript\javascript.png)

Example:

- scripts with an async attribute will execute as soon as the download is complete. This blocks the page and does not guarantee any specific execution order.

```
<script async src="js/vendor/jquery.js"></script>

<script async src="js/script2.js"></script>

<script async src="js/script3.js"></script>
```

You can't rely on the order the scripts will load in. jquery.js may load before or after script2.js and script3.js and if this is the case, any functions in those scripts depending on jquery will produce an error because jquery will not be defined at the time the script runs. 'async' should be used when you have a bunch of background scripts to load in, and you just want to get them in place as soon as possible.

- scripts with a defer attribute will load in the order they are in and will only execute once everything has finished loading. Only use this when your scripts should be run immediately and they don't have any

dependencies. Scripts loaded using the defer attribute (see below) will run in the order they appear in the page and execute them as soon as the script and content are downloaded:

```
<script defer src="js/vendor/jquery.js"></script>

<script defer src="js/script2.js"></script>

<script defer src="js/script3.js"></script>
```

In the second example, we can be sure that jquery.js will load before script2.js and script3.js and that script2.js will load before script3.js. They won't run until the page content has all loaded, which is useful if your scripts depend on the DOM being in place (e.g. they modify one or more elements on the page).

A first splash into JavaScript?

Guess the number game

Generate a random number between 1 and 100. Record the turn number the player is on. Start it on 1. Provide the player with a way to guess what the number is. Once a guess has been submitted first record it somewhere so the user can see their previous guesses. Next, check whether it is the correct number. If it is correct: Display congratulations message. Stop the player from being able to enter more guesses (this would mess the game up). Display control allowing the player to restart the game. If it is wrong and the player has turns left: Tell the player they are wrong and whether their guess was too high or too low. Allow them to enter another guess. Increment the turn number by 1. If it is wrong and the player has no turns left: Tell the player it is game over. Stop the player from being able to enter more guesses (this would mess the game up). Display control allowing the player to restart the game. Once the game restarts, make sure the game logic and UI are completely reset, then go back to step 1.

Add variables to store out data

```
let randomNumber = Math.floor(Math.random() * 100) + 1;

const guesses = document.querySelector(".guesses");
const lastResult = document.querySelector(".lastResult");
const lowOrHi = document.querySelector(".lowOrHi");

const guessSubmit = document.querySelector(".guessSubmit");
const guessField = document.querySelector(".guessField");

let guessCount = 1;
let resetButton;
```

Functions

```

```

```
function checkGuess() {
  alert("I am a placeholder");
}
```

Operators

Normal operators like +, -, *,/, >, < Shortcut operators like +=

Character	Entity
strict equal	===
non-equal	!==

You can also declare strings using backticks (`). Strings declared like this are called template literals and have some special properties. In particular, you can embed other variables or even expressions in them:

```
function checkGuess() {
  const greeting = `Hello ${name}`;
}
```

Conditionals

```
function checkGuess() {
  const userGuess = Number(guessField.value);
  if (guessCount === 1) {
    guesses.textContent = "Previous guesses:";
  }
  guesses.textContent = `${guesses.textContent} ${userGuess}`;

  if (userGuess === randomNumber) {
    lastResult.textContent = "Congratulations! You got it right!";
    lastResult.style.backgroundColor = "green";
    lowOrHi.textContent = "";
    setGameOver();
  } else if (guessCount === 10) {
    lastResult.textContent = "!!!GAME OVER!!!";
    lowOrHi.textContent = "";
    setGameOver();
  } else {
    lastResult.textContent = "Wrong!";
    lastResult.style.backgroundColor = "red";
    if (userGuess < randomNumber) {
      lowOrHi.textContent = "Last guess was too low!";
    } else if (userGuess > randomNumber) {
      lowOrHi.textContent = "Last guess was too high!";
    }
  }
```

```
  guessCount++;
  guessField.value = "";
  guessField.focus();
}
```

- The first line declares a variable called userGuess and sets its value to the current value entered inside the text field. We also run this value through the built-in Number() constructor, just to make sure the value is definitely a number. Since we're not changing this variable, we'll declare it using const.
- Next, we encounter our first conditional code block. A conditional code block allows you to run code selectively, depending on whether a certain condition is true or not. It looks a bit like a function, but it isn't. The simplest form of conditional block starts with the keyword if, then some parentheses, then some curly braces. Inside the parentheses, we include a test. If the test returns true, we run the code inside the curly braces. If not, we don't, and move on to the next bit of code. In this case, the test is testing whether the guessCount variable is equal to 1 (i.e. whether this is the player's first go or not):

```
guessCount === 1;
```

If it is, we make the guesses paragraph's text content equal to Previous guesses:. If not, we don't.

- Next, we use a template literal to append the current userGuess value onto the end of the guesses paragraph, with a blank space in between. The next block does a few checks:
  - The first if (){ } checks whether the user's guess is equal to the randomNumber set at the top of our JavaScript. If it is, the player has guessed correctly and the game is won, so we show the player a congratulations message with a nice green color, clear the contents of the Low/High guess information box, and run a function called setGameOver(), which we'll discuss later.
  - Now we've chained another test onto the end of the last one using an else if (){ } structure. This one checks whether this turn is the user's last turn. If it is, the program does the same thing as in the previous block, except with a game over message instead of a congratulations message.
  - The final block chained onto the end of this code (the else { }) contains code that is only run if neither of the other two tests returns true (i.e. the player didn't guess right, but they have more guesses left). In this case we tell them they are wrong, then we perform another conditional test to check whether the guess was higher or lower than the answer, displaying a further message as appropriate to tell them higher or lower.
- The last three lines in the function (lines 26–28 above) get us ready for the next guess to be submitted. We add 1 to the guessCount variable so the player uses up their turn (++ is an incrementation operation — increment by 1), and empty the value out of the form text field and focus it again, ready for the next guess to be entered.

Events

Events are things that happen in the browser — a button being clicked, a page loading, a video playing, etc. — in response to which we can run blocks of code. Event listeners observe specific events and call event handlers, which are blocks of code that run in response to an event firing.

```
guessSubmit.addEventListener("click", checkGuess);
```

Here we are adding an event listener to the guessSubmit button. This is a method that takes two input values (called arguments) — the type of event we are listening out for (in this case click) as a string, and the code we want to run when the event occurs (in this case the checkGuess() function).

Finishing the game functionality

```
function setGameOver() {
  guessField.disabled = true;
  guessSubmit.disabled = true;
  resetButton = document.createElement("button");
  resetButton.textContent = "Start new game";
  document.body.append(resetButton);
  resetButton.addEventListener("click", resetGame);
}
```

- The first two lines disable the form text input and button by setting their disabled properties to true. This is necessary, because if we didn't, the user could submit more guesses after the game is over, which would mess things up.
- The next three lines generate a new `<button>`element, set its text label to "Start new game", and add it to the bottom of our existing HTML.
- The final line sets an event listener on our new button so that when it is clicked, a function called resetGame() is run.

Reset the game:

```
function resetGame() {
  guessCount = 1;

  const resetParas = document.querySelectorAll(".resultParas p");
  for (const resetPara of resetParas) {
    resetPara.textContent = "";
  }

  resetButton.parentNode.removeChild(resetButton);

  guessField.disabled = false;
  guessSubmit.disabled = false;
  guessField.value = "";
  guessField.focus();

  lastResult.style.backgroundColor = "white";

  randomNumber = Math.floor(Math.random() * 100) + 1;
}
```

Loops

To print: 'apples', 'bananas', 'cherries' out in your console.

```
const fruits = ["apples", "bananas", "cherries"];
for (const fruit of fruits) {
  console.log(fruit);
}
```

This code creates a variable containing a list of all the paragraphs inside

using the querySelectorAll() method, then it loops through each one, removing the text content of each.

```
const resetParas = document.querySelectorAll(".resultParas p");
for (const resetPara of resetParas) {
  resetPara.textContent = "";
}
```

Objects

An object is a collection of related functionality stored in a single grouping. You can create your own objects, but that is quite advanced and we won't be covering it until much later in the course. For now, we'll just briefly discuss the built-in objects that your browser contains, which allow you to do lots of useful things.

.querySelector()

We created a guessField constant that stores a reference to the text input form field in our HTML:

```
const guessField = document.querySelector(".guessField");
```

.focus()

Automatically put the text cursor into the `<input>` text field as soon as the page loads:

```
guessField.focus();
```

Variables

A variable is a container for a value, like a number we might use in a sum, or a string that we might use as part of a sentence.

In this example pressing the button runs some code. The first line pops a box up on the screen that asks the reader to enter their name, and then stores the value in a variable. The second line displays a welcome message that includes their name, taken from the variable value and the third line displays that name on the page.

```
<button id="button_A">Press me</button>
<h3 id="heading_A"></h3>
```

```
const buttonA = document.querySelector("#button_A");
const headingA = document.querySelector("#heading_A");

buttonA.onclick = () => {
  const name = prompt("What is your name?");
  alert(`Hello ${name}, nice to see you!`);
  headingA.textContent = `Welcome ${name}`;
};
```

If we didn't have variables available, we'd have to ask the reader for their name every time we needed to use it!

Declaring a variable

To use a variable, you've first got to create it — more accurately, we call this declaring the variable. To do this, we type the keyword 'let' followed by the name you want to call your variable.

Initializing a variable

Once you've declared a variable, you can initialize it with a value. You do this by typing the variable name, followed by an equals sign (=), followed by the value you want to give it.

Or combine them:

```
let myDog = "Rover";
```

Variable types

- Numbers -Strings -Booleans -Arrays Once these arrays are defined, you can access each value by their location within the array: myNameArray[0]
- Objects: let dog = { name: "Spot", breed: "Dalmatian" };

Dynamic typing

JavaScript is a "dynamically typed language", which means that, unlike some other languages, you don't need to specify what data type a variable will contain (numbers, strings, arrays, etc.).

Constants

As well as variables, you can declare constants. These are like variables, except that:

you must initialize them when you declare them you can't assign them a new value after you've initialized them.

Similarly, with let you can initialize a constant, and then assign it a new value (this is also called reassigning the constant):

```
let count = 1;
count = 2;
```

using 'const', you cannot reassign

```
const count = 1;
count = 2;
```

Numbers and Operators

to check the datatype of a variable: When string instead of number: Number()

```
typeof myInt;
```

To round your number to a fixed number of decimal places, use the toFixed()

```
const lotsOfDecimal = 1.766584958675746364;
lotsOfDecimal;
const twoDecimalPlaces = lotsOfDecimal.toFixed(2);
twoDecimalPlaces;
```

Character	Entity
remainder	%
exponent	**

Sometimes you'll want to repeatedly add or subtract one to or from a numeric variable value. This can be conveniently done using the increment (++) and decrement (--) operators.

```
guessCount++;
```

Example from button that starts/stops a machine:

```
<button>Start machine</button>
<p>The machine is stopped.</p>
```

```
const btn = document.querySelector("button");
const txt = document.querySelector("p");

btn.addEventListener("click", updateBtn);

function updateBtn() {
  if (btn.textContent === "Start machine") {
    btn.textContent = "Stop machine";
    txt.textContent = "The machine has started!";
  } else {
    btn.textContent = "Start machine";
    txt.textContent = "The machine is stopped.";
  }
}
```

Handling text - strings

Declaring a string:

```
const string = "The revolution will not be televised.";
console.log(string);
```

Inside a template literal, you can wrap JavaScript variables or expressions inside ${ }, and the result will be included in the string:

```
const name = "Chris";
const greeting = `Hello, ${name}`;
console.log(greeting); // "Hello, Chris"
```

You can use the same technique to join together two variables: concatenate using ${}:

```
const one = "Hello, ";
const two = "how are you?";
const joined = `${one}${two}`;
console.log(joined); // "Hello, how are you?"
```

When button pressed 'hello {name}, nice to see you!' printed:

```
<button>Press me</button>
<div id="greeting"></div>
```

```
const button = document.querySelector("button");

function greet() {
  const name = prompt("What is your name?");
  const greeting = document.querySelector("#greeting");
  greeting.textContent = `Hello ${name}, nice to see you!`;
}

button.addEventListener("click", greet);
```

You can use ${} only with template literals, not normal strings. You can concatenate normal strings using the + operator:

```
const greeting = "Hello";
const name = "Chris";
console.log(greeting + ", " + name); // "Hello, Chris"
```

You can include JavaScript expressions in template literals, as well as just variables, and the results will be included in the result:

```
const song = "Fight the Youth";
const score = 9;
const highestScore = 10;
const output = `I like the song ${song}. I gave it a score of ${
  (score / highestScore) * 100
}%.`;
console.log(output); // "I like the song Fight the Youth. I gave it a score of
90%."
```

multiline strings:

```
const newline = "One day you finally knew\nwhat you had to do, and began,";
console.log(newline);

/*
One day you finally knew
what you had to do, and began,
*/
```

Including quotes in strings: use one of the other characters to declare the string. Or by putting a backslash just before the character: You can use the same technique to insert other special characters

```
const goodQuotes1 = 'She said "I think so!"';
const goodQuotes2 = `She said "I'm not going in there!"`;
```

Useful string methods:

Finding the length of a string:

```
const browserType = "mozilla";
browserType.length;
```

Retrieving a specific strinh character:

```
browserType[0]
```

To retrieve the last character of a string:µ

```
browserType[browserType.length - 1];
```

Testing if a string contains a substring: Often you'll want to know if a string starts or ends with a particular substring: startsWith() and endsWith() instead of includes().

```
const browserType = "mozilla";

if (browserType.includes("zilla")) {
  console.log("Found zilla!");
} else {
  console.log("No zilla here!");
}
```

Finding position of a substring:

```
const tagline = "MDN - Resources for developers, by developers";
console.log(tagline.indexOf("developers")); // 20
```

```
console.log(tagline.indexOf("x")); // -1
```

returns -1 because the character x is not present in the string.

Extract a substring from a string: If you know that you want to extract all of the remaining characters in a string after a certain character, you don't have to include the second parameter.

```
const browserType = "mozilla";
console.log(browserType.slice(1, 4)); // "ozi"
```

Changing case: The string methods toLowerCase() and toUpperCase() take a string and convert all the characters to lower- or uppercase, respectively.

```
const radData = "My NaMe Is MuD";
console.log(radData.toLowerCase());
console.log(radData.toUpperCase());
```

Updating parts of a string: Be aware that replace() in this form only changes the first occurrence of the substring. If you want to change all occurrences, you can use replaceAll().

```
const browserType = "mozilla";
const updated = browserType.replace("moz", "van");

console.log(updated); // "vanilla"
console.log(browserType); // "mozilla"
```

Arrays

Arrays consist of square brackets and items that are separated by commas. We can store various data types — strings, numbers, objects, and even other arrays. Finding the length of an array and accessing an element, finding the index of items works the same as with a string described above.

Adding items:

```
const cities = ["Manchester", "Liverpool"];
cities.push("Cardiff");
console.log(cities); // [ "Manchester", "Liverpool", "Cardiff" ]
cities.push("Bradford", "Brighton");
console.log(cities); // [ "Manchester", "Liverpool", "Cardiff", "Bradford",
"Brighton" ]
```

The new length of the array is returned when the method call completes. If you wanted to store the new array length in a variable, you could do something like this:

```
const cities = ["Manchester", "Liverpool"];
const newLength = cities.push("Bristol");
console.log(cities); // [ "Manchester", "Liverpool", "Bristol" ]
console.log(newLength); // 3
```

To add an item to the start of the array, use unshift():

```
const cities = ["Manchester", "Liverpool"];
cities.unshift("Edinburgh");
console.log(cities); // [ "Edinburgh", "Manchester", "Liverpool" ]
```

To remove the last item from the array, use pop():

```
const cities = ["Manchester", "Liverpool"];
cities.pop();
console.log(cities); // [ "Manchester" ]
```

to save removed item:

```
const cities = ["Manchester", "Liverpool"];
const removedCity = cities.pop();
console.log(removedCity); // "Liverpool"
```

To remove the first item from an array, use shift():

```
const cities = ["Manchester", "Liverpool"];
cities.shift();
console.log(cities); // [ "Liverpool" ]
```

If you know the index of an item, you can remove it from the array using splice():

```
const cities = ["Manchester", "Liverpool", "Edinburgh", "Carlisle"];
const index = cities.indexOf("Liverpool");
if (index !== -1) {
  cities.splice(index, 1);
}
console.log(cities); // [ "Manchester", "Edinburgh", "Carlisle" ]
```

In this call to splice(), the first argument says where to start removing items, and the second argument says how many items should be removed. So you can remove more than one item:

```
const cities = ["Manchester", "Liverpool", "Edinburgh", "Carlisle"];
const index = cities.indexOf("Liverpool");
if (index !== -1) {
  cities.splice(index, 2);
}
console.log(cities); // [ "Manchester", "Carlisle" ]
```

Access every item in the array, leaving you with an array containing the changed items. You can do this using map(). The code below takes an array of numbers and doubles each number:

```
function double(number) {
  return number * 2;
}
const numbers = [5, 2, 7, 6];
const doubled = numbers.map(double);
console.log(doubled); // [ 10, 4, 14, 12 ]
```

We give a function to the map(), and map() calls the function once for each item in the array, passing in the item. It then adds the return value from each function call to a new array, and finally returns the new array.

Sometimes you'll want to create a new array containing only the items in the original array that match some test. You can do that using filter(). The code below takes an array of strings and returns an array containing just the strings that are greater than 8 characters long:

```
function isLong(city) {
  return city.length > 8;
}
const cities = ["London", "Liverpool", "Totnes", "Edinburgh"];
const longer = cities.filter(isLong);
console.log(longer); // [ "Liverpool", "Edinburgh" ]
```