2. DOM & JavaScript

Design and development of web games (VJ1217), *Universitat Jaume I*

Estimated duration: 3 hours (+ 4 hours of exercises)

> "Theory is when you know something, but it doesn't work. Practice is when something works, but you don't know why. Programmers combine theory and practice: Nothing works and they don't know why." - Unknown (@CodeWisdom)

Thanks to the HTML Document Object Model[1], DOM in short, web programmers and developers are capable of accessing, changing and removing any element of an HTML document. This can be achieved by developing programs in client-side programming languages, such as the JavaScript programming language. JavaScript (JS) is a high-level, dynamic, untyped, and interpreted run-time programming language. It has been standardised in the ECMAScript language specification, thus being closely related to other ECMAScript-like languages such as ActionScript (AS). JS can access the HTML tags, properties, and contents through DOM and is able to dynamically manage all of them at runtime. In this lab session, you will be introduced to some basic concepts of JS and DOM, and will combine both for developing basic animations and event handling projects.

Contents

[1] https://en.wikipedia.org/wiki/Document_Object_Model

1 What is the HTML DOM

The HTML Document Object Model, DOM henceforth, is a standard object model and programming interface for HTML. It is a tree of objects created by the browser when it loads a web page, and its purpose is to allow JavaScript (or other programming languages) to access, change, add and delete any element of an HTML document, as well as its properties, methods and events.

1.1 DOM Tree of Objects

As the following code fragment and Fig. 1 should illustrate, the DOM is primarily an object model structured as a *tree of objects*.

```
<!DOCTYPE html>
<html>
    <head>
        <title>HTML DOM</title>
    </head>
    <body onclick="this.innerHTML = '<p>DONE</p>'">
        <div id="idx"
            style="position: absolute; left: 50px">
            <p style="color: red">CLICK ME!</p>
        </div>
    </body>
</html>
```

Every HTML, CSS, and even JavaScript, element included in a collection of documents building up a web page displayed by a browser is represented as a node (object) in a tree such as that shown in Fig. 1.

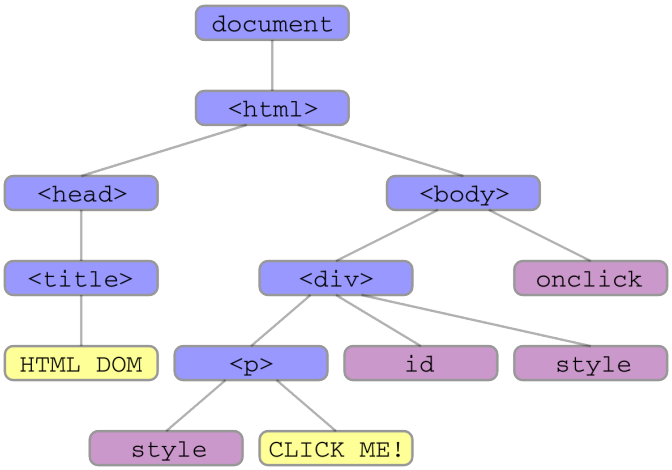

Figure 1: *Tree of objects for the HTML code in Sect. 1.1.*

1.2 DOM Programming Interface

A *programming interface* wrapping such a model, the tree of objects, is provided by the DOM to access and manage all the elements stored in the tree, taking into account their different nature: HTML/CSS elements, properties (specifying features of the objects), methods (defining executable actions on the objects), events (notifying object or environment changes)...

Any item of an HTML document (node of a DOM tree) can be accesed through the predefined `document` object in JavaScript, which offers methods to find concrete elements in the DOM, such as `getElementById()`, `getElementsByClassName()` or `getElementsByTagName()`, or properties to directly get selected elements, such as `body` or `title`.

```
<script>
    document.title = 'DOM Programming Interface';
    document.getElementsByTagName("p")[0].innerHTML =
        document.body.nodeName;
    let x = document.getElementById("idx");
    x.innerHTML = '<h1>New text</h1>' + x.innerHTML;
</script>
```

The `title` property sets or returns the title string of the current document, while the `body` property is an object representing the HTML section `<body>`. Every HTML element represented in the DOM has a `nodeName` property which contains its tagname in capitals. The method `getElementById()` returns the object identified by its argument, and `getElementsByTagName()` returns a collection (array-like container) composed of all the objects that match the tagname argument. Both searches run within the scope of the invoking object. JavaScript syntax issues are considered below, but the sentences of this example should be understandable.

Learn more about...

- `document.getElementById()`
- `document.getElementsByClassName()`
- `document.getElementsByTagName()`

1.3 Connecting DOM with CSS

DOM also allows for accessing and changing the style of HTML elements, i.e., their CSS properties. Every HTML element (object of the DOM) provides an `style` property, which holds an object that has DOM properties for all possible CSS style properties. These DOM properties store string values, which we can set to modify any aspect of the element's style.

```
<script>
    let w = document.getElementById("idx");
    w.style.left = '200px';
    w.style.backgroundColor = 'lightgreen';
    let y = document.getElementsByTagName("p")[0];
    y.style.color = 'purple';
    let z = document.getElementsByTagName("h1")[0];
    z.style.fontFamily = 'courier';
</script>
```

Some CSS style property names contain dashes, such as `"background-color"`. Such names are awkward to work with in JavaScript and they should be written as `style["background-color"]`. Alternatively we can use DOM property names having their dashes removed and the letters that follow them capitalized (`style.backgroundColor`).

2 JavaScript Basics

JavaScript (JS) was developed by Brendan Eich in 1995, just to improve user interactivity for web pages in Netscape Navigator, the most popular browser at that time. Since its standardisation by Ecma International in 1997, as ECMA-262, it has been evolving to enhance web browsing functionality and it has been incorporated to many other modern browsers. ECMAScript is the official name of the language, thus being a synonym for JS. On the contrary, in spite of its similarity to the programming language named Java, they have nothing to do with each other.

In HTML, JS code must be inserted between `<script>` and `</script>` tags. Any number of scripts can be placed in an HTML document, either in the `<body>` or `<head>` sections, or even in both. Nevertheless, placing them at the bottom of the `<body>` element is highly recommended to speed up web pages display.

To ease code reusability, instead of embedding scripts in HTML files, JS code can also be placed in external files, whose names should end with the file extension `.js`. In this case, the name of the script file must be written in the `src` (source) attribute of a `<script>` tag:

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

Several script files can be added to the same page, using a `<script>` tag for each file. Also, note that JS code in a script file cannot contain `<script>` tags, just as `<style>` tags are not included in a CSS file.

Your turn 1

Open the project `CapShieldVsMjolnir` and all its files. Run the project in Visual Studio Code and check its functionality. To do so, right-click on the

`index.html` file and choose option "Open with Live Server". We will study its code later on this lab session. Note the structure of the site and check the references to external files in `index.html`. Also, read the code in the files `index.html` and `clashscene.css` and try to relate tags and attributes to what is shown in the browser window.

Values, Types and Operators

Seven basic types of values are provided in JS: Number, String, Boolean, Undefined, Null, Symbol, and Object, whereby the first six are categorised as Primitive (immutable values), and the last one as Complex. Literals for the first three Primitive types can be written obeying certain, more or less, common rules: Numbers are written with (`3.14159`) or without (`20`) decimals, or with scientific (exponential) notation (`3141.59e-3`); Strings are text delimited by single or double quotes (`"red"`, `'light blue'`); and, there are only two Boolean values (`true` and `false`). On the other hand, there is only one Undefined value (`undefined`, for variables that exist but have nothing assigned to them) and only one Null value (`null`, for variables that exist and have `null` intentionally assigned to them).

Obviously, data values are stored in variables, which are declared using the `let` keyword. At the same time that a variable is declared a value can be assigned to it, through the assignment (=) operator, but this is not mandatory. Many variables can be declared in only one statement, separating them by commas.

```
let colors; //  <=>  let colors = undefined;
const _Co10r$ = "rainbow";
let top = 0, left, bottom = 60, right;
bottom = false;
```

A variable just declared has no value; i.e., technically, it has the value `undefined`. JS variables have dynamic types: the same variable can hold different data types. Thus, above assignments to `bottom` are valid.

Variables declared using `const` are *constants*: their values cannot be changed once set. For this reason, every constant must be initialized on declaration. But, except for such value immutability, `const` declarations are treated like `let` declarations.

Rules for legal identifier names (variables, functions...) in JS are similar to most programming languages: a sequence of letters, digits, underscores (_), or even dollar signs ($), which doesn't start by a digit. JS code is case sensitive and use Unicode charset encoding.

Double slashes, `//`, start a comment which extends until the end of the current line. Multiple line comments are written between `/*` and `*/`.

Available operators in JS, which form expressions, are also widely shared among popular programming languages:

Arithmetic: `+ - * / %` (modulo) `++` (incr.) `--` (decr.)

Assignment: `= += -= *= /= %=`

Comparison: `== != > < >= <=`

JS operators addressing type checking

Type operators `typeof` and `instanceof` let us known about types of variables and objects. For instance, `typeof []` says `Object` but `[] instanceof Array` is `true`, since `[]` is the empty array and `Array` is a special type of `Object`.
JS specific comparison operators `===` and `!==` determine equality or difference both in value and type for their operands, in contrast to common comparison operators `==` and `!=`, that check only for equality or difference in value. For instance, `0 == "0"` is `true`, thanks to automatic type conversion rules, but `0 === "0"` is `false`.

Logical: `&&` (and) `||` (or) `!` (not)

Other JS operators are the JS specific comparison operators `===` and `!==`, the conditional ternary operator (`?:`), the type operators (`typeof` and `instanceof`), and several bitwise and bitwise-assignment operators. Operator precedence in JS can also be mostly inferred from known precedences in standard programming languages. Recall that you can always break it by means of parentheses.

Learn more about...

Operators · Assignment · Comparisons

Be aware of the permissive rules of JS type checking in expressions, specially in conditions and expressions involving numbers and strings. Frequently, you won't be notified a type mismatch but you could experience unexpected results. But sometimes, you can use it for your own benefit. For instance, based on code from our example project, next sentences change the left position of a DOM object:

```
let sh = document.getElementById("capshield");
let i = parseInt(sh.style.left.slice(0,-2));
i += 23;
sh.style.left = i + 'px';
```

The left position (style property of string type) of a `<div>` element, identified with the string `"capshield"`, is sliced (through a built-in string method) to get its digits only, then converted to the corresponding number by the JS global function `parseInt`, next incremented by `23` units as a number, and finally converted back to string where this number is concatenated to the string `'px'` and assigned as a string to the corresponding style property.

Recall that the operator `+` adds numbers but concatenates strings, and if its operands are a number and a string, the result will be a string.

Objects and Arrays

In JS, an *object literal* lets you both define and create an object in only one statement. It is a list of *properties* written as `name:value` pairs, separated by commas and delimited by curly braces. In its simplest form, an *empty object* is specified by the `{}` object literal. After creation, a

new property can be added to an existing object by simply giving it a value. The next two sentences create the object `clash` with two initial properties, `shield` and `mjolnir`, and then add a third one, `timer`, to the object.

```
let clash = {shield:undefined, mjolnir:undefined};
clash.timer = undefined;
```

The object property `shield` can then be accessed as `clash.shield`. Our example project uses the main object `clash` to represent and manage the whole scene to be animated.

Other elements in objects are the *methods*, which perform actions on objects and actually are properties storing function definitions. They, as properties, can also be added to existing objects after their creation.

We postpone the explanation of the method specification until Sect. 4, but we can introduce method invocation since certain methods are always available to all objects (by inheritance). For instance, any object can run `toString`, a method for converting itself to a string representation. To invoke this method on `clash` object, for instance, we write `clash.toString()` (parentheses are required). Moreover, methods needing arguments have to be invoked providing appropriate values for them inside the parentheses.

Objects can also be declared constants, by means of `const`. But, in such a case, only the binding to the object is immutable; i.e., we will not be able to assign a new entire object to the identifier. However, we could change any of its properties and methods, or add new ones. A `const` declaration in an object actually prevents modification of the binding and not of the value itself.

Similar to objects, an *array literal* creates an array in one statement, through a list of values (expressions), separated by commas and delimited by square brackets.

```
let colors = ["cyan","green","gray","blue"];
```

As usual, any full JS array is accessed through the array name and any array element is specified by an index number between square brackets, such as `colors[i]`. JS array indexation is zero-based, which means the first item has index 0, and an *empty array* is defined by the `[]` array literal.

The array `colors` stores, in our example project, the available color labels that will be used to change the background color in the animated scene.

JS arrays are a special type of objects. Indeed, the `typeof` operator returns `'object'` when applied to an array. Moreover, JS arrays allow for storing different item types in the same array.

JS arrays offer many built-in properties and methods, which facilitate code writing. One of such properties is `length`, which holds the actual number of array items. In our example project, the following sentence uses the `length` property of the array `colors`, to compute indexes in the allowed range.

```
const LEAP = 7;
...
cIndex = (cIndex + LEAP) % colors.length;
```

A useful built-in array method is `splice`, which adds or removes items to/from the array. Its first parameter defines the position where operations take place, that is, where new items are added or existing items are removed from. The second parameter specifies the number of elements to be removed. And the rest of the parameters provide the new elements to be added. Note that the two first parameters can be 0 and only they are mandatory. Try to understand the following sentences, extracted from our example project.

```
let i = 0;
colors.splice(i,0,"light"+colors[i]);
colors.splice(i+2,0,"dark"+colors[i+1]);
// colors === ["lightcyan","cyan","darkcyan",
//               "green","gray","blue"]
```

From an initial position `i`, the `"light"` version of the color stored in index `i` is inserted before the existing color in the array, and then its `"dark"` version is inserted after it.

Program Structure

There are no official standards for the Browser Object Model (BOM), but it is the common name often used to refer to the methods and properties for JS interactivity that modern browsers have implemented and which are (almost) the same among all browsers.

The `window` object is supported by all browsers and represents the browser's window. Then, the `document` (HTML DOM) object is provided as a property of the `window` object, that is, `window.document` is the same as `document`. Note that you cannot overwrite any global standard identifier, like `window` or `document`, using `let` or `const` declarations: you can only shadow it. For each `let` and `const` declaration in the global scope, a new binding is created in this scope but no property is added to the `window` object.

In this way, we will start the execution of our JS code based on the availability of the `window` object (BOM) rather than that of the `document` object (DOM). Waiting for the BOM is safer than only waiting for the DOM, to start running JS code, because it requires more content (images, css, scripts, etc.) to be loaded and ready.

Therefore, in our HTML/CSS/JS projects, in the main JS file, we will always provide a sentence like:

```
window.onload = entryPoint;
```

Then, a function named `entryPoint` (or whatever the name you want to employ) must also be defined, containing the starting actions of our program. In Sect. 4 we will learn how to define and call functions. Moreover, in Sect. 8 we will know the meaning of the `onload` property/method on the `window` object and, in general, on any object.

On the other hand, JS programs use semicolons to separate statements, though they are optional in most cases. To avoid thinking about which cases do require it or not, ending all statements with semicolons is highly recommended. Also, JS keywords are reserved words, and this means that they cannot be used as identifiers.

Simple debugging

Although advanced debugging strategies and tools can be used, for the time being it is worth mentioning a rudimentary yet useful one: printing messages to the browser console. For instance, we might want to know the value of the variable `energy` before the function `checkForCollisions()` is called. We might then simply write:

```
console.log("Before checkForCollisions(), energy=",energy);
```

Since error messages are also displayed in the console, it is an invaluable tool for the web (game) developer.
Within a browser, press Ctrl+Shift+I to show (or hide) the console and other developer tools.
The contents of JS objects can be easily displayed with the method `toString()`:

```
console.log(colors.toString());
```

□

You can also find it useful to display an alert box with some information. It is more disruptive than `console.log`, because it has to be explicitly closed, but this drawback can turn into an advantage when this is desired.

```
alert("There are " + stars.length + " stars");
```

3 JavaScript: control statements

As it could be expected, JS provides sentences for controlling the flow of execution. You'll find that they are typical conditional and loop structures, very similar to those in other programming languages.

Conditional execution

The typical conditional statement in JS is **if**, which let us execute different actions based on a condition (expression), delimited by parentheses and that evaluates to a Boolean value. It has an optional section **else**, and requires both blocks of dependent sentences be delimited by curly braces if they contain more than one sentence. Curly braces are optional when only containing one statement.

Look at the following example to identify all above mentioned parts. In this code, we check if an object's left position plus two times its width goes beyond other object's left position. In our example project, these sentences detect when the weapons are close enough to increase the frequency of background colors change.

```
const LONG_PERIOD = 20;
const SHORT_PERIOD = 1;
let period;
let sh = clash.shield, mj = clash.mjolnir;
if (sh.left + 2 * sh.width > mj.left)
  period = SHORT_PERIOD;
else
  period = LONG_PERIOD;
```

JS syntax allows for writing successive **else if** sections, between **if** and **else**, for managing several tests with more than two choices.

The **switch** statement provides an alternative way for selecting one between many blocks of code to be executed.

```
switch ( /*expression*/ ) {
  case /*expression 1*/ :
    // code block for expression 1
    break;
  case /*expression 2*/ :
    // code block for expression 2
    break;
  ...
  default:
    // code block default value
}
```

The **switch** expression is evaluated once. Then, its value is compared in order of appearance, by means of ===, with the values of the expressions of each **case** label. If there is a match, the associated block of code is executed. The **default** keyword specifies the code to run when there is no match. The **break** keyword stops the execution of more code and case testing inside the **switch** statement. Thus, it is not necessary within the last case.

Learn more about...

if ... else switch

Loops

JS supports different kinds of loops: the usual **for** and **while** sentences, and the less frequent **for-in** and **do-while** structures. Like conditional sentences, all their blocks of dependent sentences must be delimited by curly braces if they contain more than one sentence, but curly braces can be omited is there is only one.

Observe the following **for** example and note the three expressions inside the parentheses and separated by semicolons:

```
for (let i = 0 ; i < colors.length ; i += 3) {
  colors.splice(i,0,"light"+colors[i]);
  colors.splice(i+2,0,"dark"+colors[i+1]);
}
```

The first expression initialises variables to be used within the dependent code. It is evaluated only once, before the loop starts. The second expression is the condition for running the loop and is evaluated before every iteration. And the third one defines changes on variables after each iteration, hopefully to eventually end the loop. In our example project, this loop completes `colors` array to insert `"light"` (before) and `"dark"` (after) versions for the colors initially defined.

The `while (/*condition*/) ...` sentence loops through a block of dependent code while the condition is true. The expression for this condition must evaluate to a Boolean value and is evaluated before every iteration.

The `do ... while (/*condition*/)` sentence is a variant of the while loop which tests its condition *after* running its code block. This means that at least one iteration is guaranteed to run.

The `for (/*variable*/ in /*object*/) ...` sentence serves for looping through the properties of an object.

Learn more about...

for while

Block and scope issues

Variables and constants declared through `let` and `const` are *block-level declarations*, similar to that of C-based languages: inaccesible outside of the declaration block or function, both delimited by curly braces (see the box Hoisting). Each variable or constant is created at the spot where its declaration occurs, which means that it cannot be used before it is declared. So, if it is intended to be available to the entire block, function or global scope, then it must be declared at the beginning of the scope. On the other hand, a `let` or `const` declaration can create a new variable with the same name as an existing variable in its containing scope. This new variable shadows the existing one.

Moreover, initialisation section of a `for` loop is part of the subsequent block; i.e., a `let` declaration within it makes the declared variable available only inside the loop, and no longer accessible elsewhere once the loop is complete.

Hoisting

Variable declarations using the `var` keyword and function declarations in JavaScript are treated as if they were placed at the top of the function or global scope (if declared outside of a function), regardless of where the actual declaration occurs. This default behavior is known as *hoisting*. Because of this, functions and this kind of variables (`var`) can be used before they are declared. But note that JavaScript only hoists declarations, not initialisations.
If *hoisting* is unknown, overlooked or misunderstood by a developer, the code could run with bugs (errors). To overcome it, while working with `var`, always declare all variables at the beginning of every scope, since this matches the default behaviour of JavaScript. Alternatively, use only `let` and `const` keywords for declaring variables and constants, and work always on a more standard, block-level treatment for declarations, which is something that we strongly recommend.

4 JavaScript functions and classes

Functions allow for code reusability, that is, for defining the code once, and using it many times, probably with different arguments in order to produce different results. Classes allow for generalizing objects behaviour and, particularly in JS, their definition is closely related to function definition.

Functions

A JS function is defined or declared with the `function` keyword, followed by its name and parentheses, within which parameter names separated by commas may be included. Identifier's rules apply both to function and parameter names, and the parameters behave as local variables inside the function scope. The code to be executed by the function is placed inside curly braces. Check all of these features, except parameters, in the following example.

```
function initBackground() {
  for (let i = 0 ; i < colors.length ; i += 3) {
    colors.splice(i,0,"light"+colors[i]);
    colors.splice(i+2,0,"dark"+colors[i+1]);
  };
  let decIndex = Math.random() * colors.length;
  cIndex = Math.floor(decIndex);
  let bg=document.getElementById("battlefield");
  bg.style.backgroundColor = colors[cIndex];
};
```

This function initialises the background color in our example project. Recall the purpose of the `for`, explained in the loops section, and observe how a random integer index is generated to select a color among those available in the `colors` array. The `random()` method of the `Math` object (not a constructor) returns a random number between 0 (inclusive) and 1 (exclusive), which is then scaled to the number of colors and truncated to the lower integer closer to it.

Learn more about...

Math object and its Random method

When a function is invoked, providing arguments for its parameters if required, its code is executed. Apart from some special cases, a JS function is normally called when an explicit call statement in the code is run or an event occurs. In the following example, an explicit call statement to the above defined `initBackground()` function appears. Similarly, an explicit call to the method `initWeapons()` of the `clash` object is run. This method is defined and explained below. Later on, the invocation of a function in the context of event management will be discussed.

```
function entryPoint() {
  initBackground();
```

```
  clash.initWeapons();
  const T = 1000 / 50; // 20 ms
  clash.timer = setInterval(clash.animate,T);
};
```

Note that the function `entryPoint()` carries out some initial tasks in our example project, arranging background and weapons. In addition, the `setInterval()` function, actually a method of the object `window`, sets up a timer for the method `animate()` of the object `clash`, to be run every 20 milliseconds, that is, 50 times per second. By making `animate()` to generate and display a (possibly different) frame each time it runs, an animated scene can be shown on the browser.

Learn more about...

window.setInterval()

Thus, the method `animate()` should be declared as follows, which also illustrates that the definition of a method in JS objects is simply the assignment of the definition of an anonymous function to a property.

```
clash.animate = function () {
  clash.advanceWeapons();
  clash.changeBackgroungColor();
};
```

Tasks managed by `animate()` relate to advancing weapons and changing background color, and are independently carried out by two other `clash` methods.

Besides presenting a second example of method declaration, the code of `advanceWeapons()` highlights some important facts. First, general `advanceWeapons()` operation splits into individual micro-movements of the two items on the scene, one of them increasing and decreasing the other one. Then, after modifying their position on the scene, their eventual clashing is tested and, if it succeeds, animation is stopped by calling the `clearInterval()` function with the `timer` obtained at the beginning.

```
clash.advanceWeapons = function () {
  const SHIELD_MOVE = 1;
  const MJOLNIR_MOVE = -2;
  clash.shield.move(SHIELD_MOVE);
  clash.mjolnir.move(MJOLNIR_MOVE);
  let sh = clash.shield, mj = clash.mjolnir;
  if (sh.left + sh.width / 2 > mj.left)
    clearInterval(clash.timer);
};
```

Functions often compute a value that has to be returned back to its caller. In JS, the `return` keyword is used within the code of a function declaration to place the point for sending back the value. When the execution flow reaches a `return` statement, the function stops its own execution and the flow continues on the code following the invoking statement. Invocations to functions returning a value can be placed everywhere an expression for that type of value can be used.

Your turn 2

In `shieldhammer.js` define a function, named `randomArrayIndex`, to generate and return a random integer index within the current range of the array that this function has to receive as a parameter. To this end, use the corresponding sentences of `InitBackground()` and, after writing it, replace these sentences by an adequate call to the function.

Classes

In addition to declaring and creating single objects, JS let us declare *object types* that afterwards can be used to create many actual objects from them. The standard way to do it in ECMAScript 6 is by means of *class declaration*. Observe the subsequent declaration:

```
class Weapon {
  constructor(x) {
    this.image = x;
    let s = this.image.style.left;
    this.left = parseInt(s.slice(0,-2));
    s = this.image.style.fontSize;
    this.width = parseInt(s.slice(0,-2));
  };
  move(n) {
    this.left += n;
    this.image.style.left = this.left + "px";
  };
};
```

`Weapon` is a *class declaration*: it begins with the `class` keyword followed by the class name, a special object constructor method named `constructor` and the use of the `this` keyword in its sentences. Naming classes with an upper-case first letter is recommended, so as to match JS built-in constructors naming style. *Own properties*, those that occur in the instance (object) rather than the prototype (class), can only be created inside a class method. So, creating all own properties inside the `constructor` method is advisable. In this way, a single place in the class will be responsible for all of them. Class methods must be placed inside the class declaration; there is no direct way for adding them outside the scope of this declaration, as it was the case for objects (see the box Class-like Structures in ECMAScript 5).

Notice that `this` is not a variable: you cannot change its value. It is a keyword that acts as a placeholder for the future actual reference of any object constructed. At the object creation time, `this` will be substituted by the actual object reference. In other words, `this` represents the object that "owns" the JS code.

Within our example project, the objects built through this class will serve to model in the code the moving weapons on the scene. Thus, the DOM object (`<div>`), to which each one is connected, is passed as an argument, and properties for storing the necessary data and a method for updating it are provided. The `image` property holds an access reference to the own DOM object, required for managing the `style.left` property of the DOM object,

Class-like Structures in ECMAScript 5

ECMAScript 6, also known as ECMAScript 2015, introduced a new *syntax* for class-like structures specified in ECMAScript 5 which does not set up a new object-oriented inheritance model to JavaScript, but it is merely a syntactical renovation of the existing prototype-based inheritance.
Compare the classical syntax (below) with the new one:

```
function Weapon(x) {
  this.image = x;
  let s = this.image.style.left;
  this.left = parseInt(s.slice(0,-2));
  s = this.image.style.fontSize;
  this.width = parseInt(s.slice(0,-2));
};

Weapon.prototype.move = function (n) {
  this.left += n;
  this.image.style.left = this.left + "px";
};
```

In this code, `Weapon` is an *object constructor function* that creates three properties. The `move` method is assigned to the `Weapon` prototype, so the same function code will be shared by all objects created from the `Weapon` constructor function. This can also be used as an indirect way for adding a method outside the scope of a class declaration,
More detailed descriptions can be found in:
- MDN web docs - moz://a. JavaScript Classes
- Underst. ECMAScript 6 - Introducing JS Classes

responsible for the display location of the object in the browser. Also, integer numerical versions, `left` and `width`, of the corresponding string values in the object `style` are stored, to test appropriate conditions in other sentences of the code. The `move()` method updates the numerical own property `left` and then uses it for setting up the `style.left` property of the DOM object.

Once a class declaration is available, new objects of the same type can be created. This operation needs the use of the keyword `new`.

```
clash.initWeapons = function () {
  let sh = document.getElementById("capshield");
  clash.shield = new Weapon(sh);
  let mj = document.getElementById("mjolnir");
  clash.mjolnir = new Weapon(mj);
};
```

The method `initWeapons()` of the object `clash` handles the creation of these objects representing the weapons in the object `clash` and connecting them with the DOM objects displaying the weapons in the browser window.

Your turn 3

Copy the whole project `CapShieldVsMjolnir` to a new one named `CapShieldVsMjolnirClass`. Then, close all files from the first project and open all ones from the duplicated project. Now, in `shieldhammer.js`, declare a class, named `Clash`, able to build a new object `clash` fully equivalent to the current object `clash`. Finally, substitute in the code the current object `clash` by the new class declaration and a new object `clash` built from `Clash`.

5 OOP with JavaScript

JavaScript provides some features for some sort of object-oriented programming by means of *prototypes*, which in fact are objects automatically coupled as a default property `prototype` to each object created. Thus, for instance, the class declaration `Weapon` inherits its prototype from property `Function.prototype`, and each object created from `Weapon`, such as `clash.shield`, inherits its prototype from property `Weapon.prototype`.

Prototype relations among JavaScript objects form a tree-shaped structure, rooted in `Object.prototype`. It provides a few methods that become available in all objects, such as `toString`, which converts an object to a string representation. This kind of limited inheritance is known as *prototype-based inheritance*.

JavaScript objects can offer interfaces to communicate with their environments, with the intention of *encapsulating* the internal details that make them up. In addition, JavaScript also allows for writing code with *polymorphism*, so that different objects are developed to expose the same interface and other parts of the code work on any object having that common interface. And, moreover, some kind of *class specialization* can be carried out in JavaScript through prototype inheritance of the object constructors, allowing one type for obtaining their prototype from that of another one and then adding and overriding properties as required.

All these features help writing clearer and more organized and reusable code. You are encouraged to extend your knowledge on these issues on JavaScript. It is out of the scope of this introductory material to cover in depth these additional features. Chapter 6 of the book of M. Haverbeke is a good reading on this topic.

6 Creating nodes in the DOM

HTML DOM can be completely edited, by changing, adding or removing any kind of node. Not only elements but also attributes or contents of the elements. Every edited node has to be managed through its parent node in order to preserve the integrity of the DOM tree.

To add a new element, the element node is first created and then appended to the selected parent element. Creation of an element is carried out by the `createElement()` method of the `document` object, while the `appendChild()`

method of the parent element is in charge of hooking the new element. The required actions are like this:

```
let bg=document.getElementById("battlefield");
let div1 = document.createElement("div");
bg.appendChild(div1);
```

Adding textual content to an element node, either new or existing, must follow the same steps as before, but using the specific `createTextNode()` method of the `document` object. These actions can be written like this:

```
let letter0 = document.createTextNode("0");
div1.appendChild(letter0);
```

Moreover, to add an attribute to an element node, either new or existing, a similar sequence of actions must be followed, but the details differ. Creation of the attribute is done by the `createAttribute()` method of the `document` object, while the specific `setAttributeNode()` method of the parent element links the new attribute node to the element node. Moreover, the attribute value has to be assigned by means of the `value` property of the attribute node. These actions look like this:

```
let att = document.createAttribute("id");
att.value = "capshield";
div1.setAttributeNode(att);
```

Lots of methods and properties are available for navigating, managing, and editing the DOM elements, attributes, contents, and style objects. Becoming acquainted with them will help you develop more functional and precise JS code in HTML5 games.

Learn more about...

HTML DOM Reference

Your turn 4

Copy the whole project `CapShieldVsMjolnir` to a new one named `CapShieldVsMjolnirCreation`. Then, close all files from the first project and open all ones from the duplicated project. Now, in `index.html`, delete the inner `<div>` tags, identified as `"capshield"` and `"mjolnir"`, and all their contents. Next, in `shieldhammer.js`, introduce a call to `initMyDOM();` as the first sentence in the definition of the function `entryPoint()`, and then declare the function `initMyDOM()`. This function has to create all the HTML elements, attributes and contents just deleted in `index.html`. Finally, test the resulting project to check that both approaches are functionally equivalent.

7 Animating in the *canvas*

In the first lab session we learnt how to use the **canvas** DOM element, which provides a programming interface for drawing shapes or embedding images onto a raster picture area. Each item painted on the canvas is immediately converted to coloured pixels on a raster. Thus, the canvas does not retain any previously known feature of the shape or image drawn, such as the position or the contour. To handle geometrical information on the items drawn on the canvas, these data must be kept apart, in an associated data structure in the code. Therefore, if we wanted to move any item drawn on the canvas, we would have to clear (erase) the area of the canvas which includes the item and redraw it into a new position.

To illustrate the process of displaying an animation on a `<canvas>` element, we are going to replace the two `<div>` elements including the letters `0` and `T`, in our example project, with two images which will be drawn on the `<canvas>`.

Your turn 5

Copy the whole project `CapShieldVsMjolnir` to a new one named `CapShieldVsMjolnirCanvas`. Then, close all files from the first project and open all ones from the duplicated project. Now, in `index.html`, substitute the code within the `<body>` element by that of the first code fragment shown below, and in `shieldhammer.js`, sustitute the complete definition of the `clash.initWeapons()` and `clash.advanceWeapons()` methods and of the `Weapon` class declaration by those of the remaining code fragments shown in this section. Finally, test the resulting project to see the new animation displayed in the browser.

In `index.html`, the `<div>` elements identified by `"capshield"` and `"mjolnir"` are replaced with a `<canvas>` element and it gets an `id` attribute so that it can be later accessed from JS code.

```
<div id="battlefield">
    <canvas width="800" height="600" id="scene">
        Your browser does not support HTML
        Canvas. Please shift to another browser.
    </canvas>
</div>
```

Afterwards, in the method `initWeapons()` of the main `clash` object, the coordinator of the animation, we get the object representing the `<canvas>` element from the DOM and, from this element, the required 2D context providing a drawing interface. This context is passed to each weapon constructor along with its corresponding image filename so that it can be responsible for its own displaying. Also, the initial position of its image on the `<canvas>` is passed to each weapon constructor.

```
clash.initWeapons = function () {
  clash.cv = document.getElementById("scene");
  let ctx = clash.cv.getContext("2d");
  let sh = "images/CaptainAmericaShield.png";
  clash.shield = new Weapon(ctx,sh,0,260);
  let mj = "images/ThorMjolnir.png";
  clash.mjolnir = new Weapon(ctx,mj,651,250);
};
```

Observe that two objects are created from the `Weapon` class: `clash.shield` and `clash.mjolnir`. Each one stores the 2D context and, also, the left and top coordinates used for placing the image on the `<canvas>`. An image object, stored in the property `image`, is created by the `Image()` constructor, which is functionally equivalent to `document.createElement('img')`. This object is in charge of the bitmap which depicts the assigned weapon. Thus, its `src` attribute points to the file which contains the image and its `onload()` method is coded to run the first drawing of the weapon bitmap on the canvas. This method runs as soon as the image bitmap is loaded in main memory (more details will be given in next section).

In addition, within the `onload()` code, the `width` property of the `Weapon` object is set from the attribute `naturalWidth`, which is read from the image file. This assignment is required if we want the methods `advanceWeapons()` and `changeBackgroungColor()` of the `clash` object to still work, since `width` is a property of `Weapon` instances that is used in both methods. But getting this assignment to work, requires first recording in the `myparent` ad hoc property of the `image` object a reference for its parent `Weapon` object, and then waiting for the image bitmap to be fully loaded to take the effective value of the `naturalWidth` property.

Note that the `this` keyword appearing in the scope of the `Weapon` class declaration refers to a different object than the `this` keyword written in the scope of the `onload` method for the `image` object. Take your time and analyse the following code for `Weapon` class declaration, in order to grasp all ideas just introduced.

```
class Weapon {
  constructor(c2d,filename,l,t) {
    this.canvas2d = c2d;
    this.image = new Image();
    this.image.myparent = this;
    this.image.src = filename;
    this.image.onload = function () {
      let m = this.myparent;
      m.canvas2d.drawImage(this,l,t);
      m.width = this.naturalWidth;
    };
    this.left = l;
    this.top = t;
    this.width;
  };
  clear() {
    let l = this.left, t = this.top;
    let w = this.image.naturalWidth;
    let h = this.image.naturalHeight;
    this.canvas2d.clearRect(l,t,w,h);
  };
  move(n) {
    this.left += n;
    let i = this.image;
    let l = this.left;
    let t = this.top;
    this.canvas2d.drawImage(i,l,t);
  };
};
```

Event Objects

An argument can be passed to an event handler function: the event object, which provides additional information about the event, such as what were the mouse coordinates or which key was pressed. A lot of handlers actions require managing event objects.
JAVASCRIPT.INFO Event objects

Once the image data for `Weapon` objects are loaded, the animation starts and their `move()` and `clear()` methods do the image micro-movements. These methods use the canvas methods `drawImage()` and `clearRect()` for painting and cleaning the canvas, respectively. The `move()` method parameter sets up how many pixels each weapon has to advance to the left or right.

Read above the details of the code for these methods, and below the sequence of calls to them. After movements are done, weapons' new positions are tested to detect their "clash" instant. When the test succeeds, the animation stops.

```
clash.advanceWeapons = function () {
  const SHIELD_MOVE = 1;
  const MJOLNIR_MOVE = -2;
  clash.shield.clear();
  clash.mjolnir.clear();
  clash.shield.move(SHIELD_MOVE);
  clash.mjolnir.move(MJOLNIR_MOVE);
  let sh = clash.shield, mj = clash.mjolnir;
  if (sh.left + sh.width / 2 > mj.left)
    clearInterval(clash.timer);
};
```

Your turn 6

Change the order of the four sentences after the `const` declarations of the `clash.advanceWeapons()` method, like this:

```
clash.shield.clear();
clash.shield.move(1);
clash.mjolnir.clear();
clash.mjolnir.move(-2);
```

Then, run again the animation and analyse the differences. Can you explain what is happening to the weapon images final display?

Learn more about...

HTML5 Canvas Tutorial and Reference

8 Events and their management

A user click, a timer timeout, or a key press are examples of the many *events* that can happen in any interactive (graphical) program. In plain words, an event is "something that

Event Propagation

By default, an event ocurring in a DOM or BOM element is propagated to its ancestors in the tree (containers, in terms of HTML tags), and all of them have a chance for handling the event. Propagation default behaviour can be modified or stopped, and every propagated handler can determine where the event actually happened.

↗ JAVASCRIPT.INFO Bubbling and capturing

happens". Unlike pure sequential programming, where it is clear in which order things occur, events in an interactive environment arrive in arbitrary moments. For instance, we do not know when a user will click a button, but whenever this happens, the program should properly respond to that click. A different programming paradigm, *event-driven programming*, is required to develop this kind of programs. In event-driven programming, we should basically say two things:

- which event we are interested in (*event registration*), and
- how the program should respond to that event (*defining the event handler*).

HTML DOM and BOM in browsers provide mechanisms to deal with both requirements, which allow JS programmers to register functions as handlers for specific events.

Given an event, such as `"load"`, `"click"`, `"mouseover"` or `"keypress"`, each DOM and BOM element provides an attribute, named `"onload"`, `"onclick"`, `"onmouseover"` or `"onkeypress"`, ready for registering a handler function to attend the event, every time it occurs. We have already introduced one of these attributes:

```
window.onload = entryPoint;
```

The `"load"` event in the `window` object is normally fired when the entire page has been loaded, including its content (images, css, scripts, etc.). Thus, in our HTML/CSS/JS projects, we want the code to start running when this event is generated in the browser.

Such event attributes, in DOM and BOM elements, can register only one handler per node. But sometimes more than one handler have to be registered in a given node. For this purpose, the `addEventListener()` method allows to add any number of handlers per node, and is also available to every DOM and BOM element.

Afterwards, handlers registered for an event can be removed through the method `removeEventListener()`, also available in every DOM and BOM element. It is called with arguments similar to those used in `addEventListener()`, but requiring the handler function to have a name, the same name that must be used to register it for the first time.

The following code illustrates how to start and stop the animation we are dealing with, based on detecting when the mouse pointer is over or out of the canvas. In the `entryPoint()` function, the `mouseOverOutCanvas()` method of `clash` object is run to register handlers for events `"mouseover"` and `"mouseout"` in the `cv` attribute (canvas object) of `clash`. Note the differencies when the `onmouseover` attribute or the `addEventListener()` method is used.

```
function entryPoint() {
  initBackground();
  clash.initWeapons();
  clash.mouseOverOutCanvas();
  const T = 1000 / 50; // 20 ms
  clash.timer = setInterval(clash.animate,T);
};

clash.mouseOverOutCanvas = function () {
  clash.cv.onmouseover = function () {
    clash.advancing = true;
  };
  clash.cv.addEventListener("mouseout",
                            function() {
    clash.advancing = false;
  });
};
```

The handlers for these mouse events, over or out of the canvas, simply set a Boolean attribute of the `clash` object. It has to be initially defined in the object declaration and, at any moment, is tested whether o not to allow the weapons advance.

```
clash.advancing = false;

clash.animate = function () {
  if (clash.advancing)
    clash.advanceWeapons();
  clash.changeBackgroungColor();
};
```

Your turn 7

In the file `shieldhammer.js` of the project `CapShieldVsMjolnirCanvas`, replace the code of the function `entryPoint()` and the method `clash.animate()` with the code shown above in this section. Also, introduce in the JS file the new code for `clash` written above: the declaration and initialisation of the `advancing` property and the definition of the method `mouseOverOutCanvas()`. Finally, test the resulting project to see the interaction between the event registration and handling and the animation.

9 Final example: changing screens

We conclude our introduction to JS and HTML DOM by showing how to deal with several screens. This second project also reinforces and extends already covered content such as canvas animation or event handling, though new details deserve some attention.

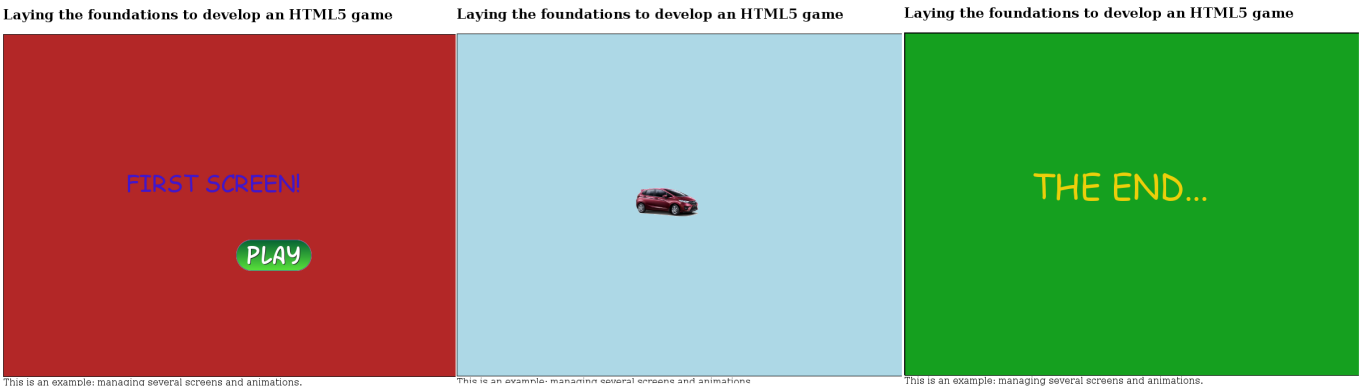

Figure 2: *`ScreenCarTravel` project deals with several screens and animates a car.*

Your turn 8

Open the project `ScreenCarTravel` and all of its files. Run the project and test its functionality before checking the code in the HTML, CSS, and JS files. By clicking on the *Play* button, you will see the sequence of screens shown in Figure 2. You can move or stop the car in the second screen by clicking on it. If the car is moving then it will stop; otherwise, it will move. The third screen is shown when the car moves beyond the right boundary of the canvas.

In `index.html` there are two `<div>` elements that are used to "store" both the start and end screens —employing the identifiers `gamestartscreen` and `endingscreen`, respectively— and a `<canvas>` element to draw the car and perform the animation in the second screen.

```
<div id="gamecontainer">
    <canvas width="800" height="600"
        id="gamecanvas" class="gamelayer">
        Your browser does not support HTML5 Canvas.
        Please shift to another browser.
    </canvas>
    <div id="gamestartscreen" class="gamelayer">
        <img src="images/play.png" alt="Play Game"
             onclick="myGame.showCanvas()">
    </div>
    <div id="endingscreen" class="gamelayer">
    </div>
</div>
```

Note that the same class has been defined for all of these HTML tags —`gamelayer`— so that all of them can be handled at once if required. Also, observe that a call to a `myGame.showCanvas()` method has been included in the `onclick` attribute of the *Play* button's image —an `<img>` tag in HTML has been used.

The file `screens.css` sets the dimensions of the stage, the position of all the screens, the background images, the position of the *Play* button, and the background colour of the canvas. Note the use of the attribute `cursor` to specify the form that the mouse cursor will take whenever it is placed on top of the image.

In file `carmoves.js`, we want to draw your attention to the methods `myGame.init()`, `myGame.showCanvas()` and `myGame.showEnd()`, where transitions between screens are coded by first hiding all of them and then showing the proper one. This is carried out by means of a value assigned to the `style.display` property, `'none'` for hiding and `'inline'` for showing.

The second point of interest in this file is the `"mousedown"` handler function. Note the event object passed and the conversion of the mouse pointer location from browser to canvas coordinates.

The rest of the code should be understandable recalling all the issues we have addressed through this lab session.

References

To prepare this document we have found inspiration in, basically, these books:

- Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*, Third Edition, https://creativecommons.org/licenses/by-nc/3.0/, 2018
- David Flanagan. *JavaScript: The Definitive Guide*, Sixth Edition, O'Reilly Media, Inc., 2011
- Nicholas C. Zakas. *Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers*, No Starch Press, First Edition (September 3, 2016) http://freecomputerbooks.com/Understanding-ECMAScript-6.html#downloadLinks; Leanpub, 2015

A good place to starting out in JavaScript and HTML DOM is `https://www.w3schools.com`:

- JavaScript Tutorial
- JavaScript HTML DOM.
- JavaScript and HTML DOM Reference.

10 Exercises

1. Define first a `800x600px` `<canvas>` and then a JS/-DOM function for drawing a *Pile of Poo emoji* similar to that shown in Fig. 3, which has been actually developed through canvas methods. Your function should:

Figure 3: *Pile of Poo emoji drawn by a JS/DOM function on the canvas.*

   - declare as parameters: a canvas 2D context, and x and y coordinates of a reference point for drawing;
   - consider (x, y) the top left point of a bounding box for the picture;
   - the position of all the elements of the picture should refer to the x and y coordinates, in such a way that changing the values of x and y result only in a shift of the picture on the canvas.

2. Define an animation for the *Pile of Poo emoji* in such a way that it (actually its assumed bounding box) makes a walk from the bottom left corner up to the top right one of the canvas. Use the function developed before, with appropriate arguments, to do the animation.

3. Develop a JS/DOM project to implement a *die of six screens* whose transitions are restricted to their adjacency in the die and managed by pressing the four arrow keys. Take also into account the following guidelines:
   - Combine HTML, CSS and JS files to structure the actions and appearance.
   - Define an `800x600px` `<div>` background and another `<div>` for each one of the six screens.
   - In each screen include the proper image, among the six images given: `die[123456].png`.
   - Position each image in the center of the scene and display it in natural width.
   - Define a keyboard event handler for managing key presses, dealing only with the four arrow keys. This handler *must be registered* for the `document` object and should check for the `"keydown"` event.