UNIT 1

JAVASCRIPT

**Part 2
BOM and DOM. Events.
Global functions. Dates.
Regular expressions.**

Client-side Web Development
2nd course – DAW
IES San Vicente 2018/2019
Author: Arturo Bernal Mayordomo

Index

Working with the BOM and DOM

Usually, when programming with JavaScript, we're inside a browser context. In this context we have some built-in objects (and functions) that we can use in order to interact with the browser and the HTML document.

Browser Object Model (BOM)

window object

The **window** object represents the browser and it's the main object when we create web pages. Everything is contained inside this window object (global variables, the document object, location object, navigation object, …). The window object can be **omitted**, accessing all its properties directly.

Check all window's (and other global objects) methods and properties here:

window object, location object, document object, history object, navigator object, screen object.

Examples:

```javascript
'use strict';
// Window's full width and height (excludes Window Manager's upper decoration bar)
console.log(window.outerWidth + " - " + window.outerHeight);

// Opens a new window (returns a reference to that window so you can close it from here).
window.open("https://www.google.com");

// screen object properties
console.log(window.screen.width + " - " + window.screen.height); // Screen width and height (screen resolution)
console.log(window.screen.availWidth + " - " + window.screen.availHeight); // Excludes Operating System bars

// navigator object properties
console.log(window.navigator.userAgent); // Prints browser information
window.navigator.geolocation.getCurrentPosition(function(position) {
    console.log("Latitude: " + position.coords.latitude + ", Longitude: " + position.coords.longitude);
});

// Global variables
let glob = "Hello";
console.log(window.glob); // Prints "Hello". Global variables are stored in the window object.

// Like global variables, we can omit the window object (it's implicit)
console.log(history.length); // Number of pages in history. Same as window.history.length
```

Timers

There are two kind of timers we can create in JavaScript in order to execute some code asynchronously in the future (specified in milliseconds), **timeouts** and **intervals**. The first one executes its code only once (we must create another manually if we want to repeat something), and the second one repeats itself every X milliseconds forever (or until we cancel it).

- **timeout(function, milliseconds)** → function is called after milliseconds

```
console.log(new Date().toString()); // Prints current date
setTimeout(() => console.log(new Date().toString()), 5000); // Will execute the function in 5 seconds (5000 milliseconds)
```

- **cleartimeout(timeoutId)** → Cancels a timeout (before it's called)

```
// setTimeout returns a number with its id, so we can cancel it
let idTime = setTimeout(function() {
    console.log(new Date().toString());
}, 5000);
clearTimeout(idTime); // Cancels the timeout
```

- **setInterval(function, milliseconds)** → The difference with timeout is that when the timer finishes and executes, it resets and repeats every X milliseconds automatically until we cancel it.

```
let num = 1;
// Prints a number and increments it every second
setInterval(function() {
    console.log(num++);
}, 1000);
```

- **clearInterval(intervalId)** → Cancels an interval (won't repeat anymore).

```
let num = 1;
let idInterval = setInterval(function() {
    console.log(num++);
    if(num > 10) { // When we print 10, we stop the timer from repeating anymore
        clearInterval(idInterval);
    }
}, 1000);
```

- **setInterval/setTimeOut(functionName, milliseconds, arguments…)** → We can specify an existent function name. If it requires parameters, we can pass them to the function after we specify the milliseconds.

```
function multiply(num1, num2) {
    console.log(num1 * num2);
}

setTimeout(multiply, 3000, 5, 7); // After 3 seconds it will print 35 → multiply(5, 7)
```

location object

The **location** object (not to be mistaken with navigator.geolocation) contains information about the current url in the browser. We can also modify that url as well, using this object.

```
console.log(location.href); // Prints current URL
console.log(location.host); // Prints server's hostname (or ip) like "localhost" or "192.168.0.34"
console.log(location.port); // Prints server's port number (usually 80)
console.log(location.protocol); // Prints used protocol (http or https)

location.reload(); // Reloads the current page
location.assign("https://google.com"); // Loads a new page. The parameter is the new url
location.replace("https://google.com"); // Loads a new page replacing current one in the history object
```

To navigate through pages we have visited in the current window or tab, we can use the **history** object. This object has a few useful methods:

```
console.log(history.length); // Prints the number of pages stored

history.back(); // Goes back to the previous page
history.forward(); // Goes forward to the next page
history.go(2); // Go fordward 2 pages (-2 would be go backwards)
```

What if we don't want to reload the document when we go back and forward in the history?. For example, our page is controlled by JavaScript methods and we want to undo or redo things when a user presses the back or the forward button.

In order to do that we should use **history.pushState()**. This method takes an JSON object as first parameter with information about changes, for example, and the title (usually ignored) as the second parameter. We could use **replaceState()** if we don't want a new entry to be inserted in the history.

In our web page the **window.onpopstate** property should be assigned to a function. This function will be called every time we go back or forward in the history object (navigating through the different added states). It will receive a parameter representing the event, that has a property named **state** that contains the JSON object that was assigned to the current state. We can access that state object anytime through **history.state** (the first page will have a null state).

File: example1.html

```html
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <script src="example1.js"></script>
    <button onclick="goBack()">Previous page</button>
    <button onclick="goNext()">Next page</button>
  </body>
</html>
```

File: example1.js

```javascript
'use strict';

// Event for capturing history navigation
window.onpopstate = function (event) {
   if(event.state) {
      console.log("I'm at page " + event.state.page);
   } else {
      console.log("I'm at the first page");
   }
};

let page = 1;

function goBack() {
   history.back();
}
```

```
function goNext() {
    // history.state == null if it's the first page
    let pageNum = history.state?history.state.page + 1:2;
    history.pushState({page: pageNum}, "");
    console.log("I'm at page " + pageNum);
}
```

Dialogs

In every browser, a set of dialogs to interact with the user is offered. However, those dialogs are not customizable and every browser implements them in its own way, so it's no recommended to use them in a production application. But, for testing, they're a good choice (in production we should use HTML form elements, etc.).

The **alert** dialog, shows a message (along an OK button) inside a modal window. For testing purposes we can use console.log() in the same way.

```
alert("Hello everyone!");
```
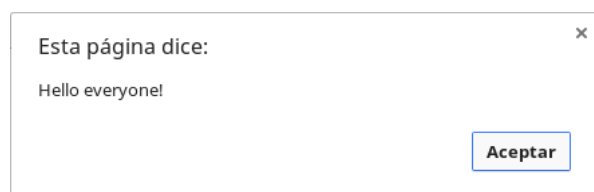
The **confirm** dialog is similar but it also returns **true** or **false**. It displays 2 buttons (cancel → false, ok → true) so the user can respond to the shown message.

```
if(confirm("Do you like dogs?")) {
    console.log("You are a good person");
} else {
    console.log("You have no soul");
}
```
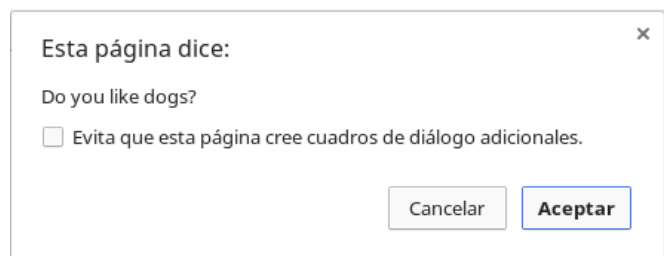
The **prompt** dialog shows an input box after the message so the user can enter some data he/she's asked for. It will return an **string** with the input's value. If the user presses **cancel** button or **closes** the dialog, it will return "null" (a string, yes...). We can set a default value as the second parameter.

```
let name = prompt("What's your name?", "Nobody");

if(name !== "null") {
    console.log("Your name is: " + name);
} else {
    console.log("You didn't answer!");
}
```
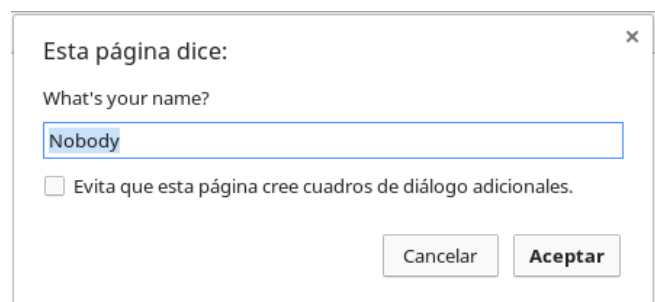
Document Object Model (DOM)

The Document Object Model is a tree structure that contains a representation of all HTML nodes including their attributes. In this tree, everything is a node, and we can

add new nodes, remove existing ones or change them.

```html
<html>
▶<head>…</head>
▼<body>
   ▼<div>
      <p>Hello</p>
      ▼<p>
         "Go to "
         <a href="https://google.es">Google</a>
      </p>
   </div>
   <script src="example1.js"></script>
</body>
</html>
```

The main object in the DOM is the **document** object. This is a global object in JavaScript (inside window like everything else). Every HTML node contained inside the document is an element object, and those elements contains attributes and style.

Manipulating the DOM using plain JavaScript is more complicated than using the help of a library like JQuery or a framework like Angular. Here we'll review some basic methods and properties of DOM elements in JavaScript.

Navigating through the DOM

- **document.documentElement** → Returns the <html> node element.

- **document.head** → Returns the <head> node element.

- **document.body** → Returns the <body> node element.

- **document.getElementById("id")** → Returns the element that has the specified id or **null** if it doesn't exist.

- **document.getElementsByClassName("class")** → Returns an array of elements that have the specified class. If you call this method from a node instead of document, it will search only inside that node.

- **document.getElementsByTagName("HTML tag")** → Returns an array of those elements that correspond to the specified HTML tag (example: "p").

- **element.childNodes** → Returns an array with the descending (child) nodes inside. It includes **text nodes** and **comment nodes**.

- **element.chidren** → Same as above but excluding text and comment nodes (only HTML nodes). Usually more recommended.

- **element.parentNode** → Returns the parent node of an element.

- **element.nextSibling** → Returns the next node at the same level (brother). The method **previousSibling** does the opposite. It's recommended to use **nextElementSibling** or **previousElementSibling** to get only HTML elements.

```html
<!DOCTYPE>

<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <ul>
        <li id="firstListElement">Element 1</li>
        <li>Element 2</li>
        <li>Element 3</li>
    </ul>
    <script src="./example1.js"></script>
  </body>
</html>
```

File: example1.js

```javascript
let firstLi = document.getElementById("firstListElement"); // Returns <li> node object

console.log(firstLi.nodeName); // Prints "LI"
// Prints 1. (element -> 1, attribute -> 2, text node -> 3, comment node -> 8)
console.log(firstLi.nodeType);
// Prints "Element 1". firstChild returns its text node
console.log(firstLi.firstChild.nodeValue);
console.log(firstLi.textContent); // Prints "Element 1". Other way to get text content

// Iterate through all the elements in the list
let liElem = firstLi;
while(liElem !== null) {
    console.log(liElem.innerText); // Prints the text inside the <li> element
    liElem = liElem.nextElementSibling; // Goes to the next <li> in the list
}

let ulElem = firstLi.parentElement; // Gets <ul> element. Similar to parentNode.
/*  Prints the HTML code inside the <ul> element:
    <li id="firstListElement">Element 1</li>
    <li>Element 2</li>
    <li>Element 3</li> */
console.log(ulElem.innerHTML);
```

Query selectors

One of the main features that JQuery introduced when it was released (back in 2006) was the possibility of getting HTML elements based on their CSS properties (class, id), attributes,…. For many years browsers have already implemented this feature natively (query selectors) without the need to use JQuery only for that.

- **document.querySelector("selector")** → Returns the first element that matches the selector.

- **document.querySelectorAll("selector")** → Returns an array with all the elements that match the selector.

Examples of CSS selectors we can use to find elements:

a → Elements with the HTML tag <a>

.class → Elements with the class "class"
#id → Element with the id "id"
.class1.class2 → Elements that have both the class "class1" **and** the class "class2"
.class1,.class2 → Elements with the class "class1" **or** elements with the class "class2"
.class1 p → Elements with the tag <p> (paragraphs) inside elements with the class "class1"
.class1 > p → Elements with the tag <p> that are immediate children of elements with class "class1"
#id + p → Element which is a paragraph <p> and is the next element (brother) of an element with id "id"
#id ~ p → Elements which are paragraphs <p> and brothers of an element with the id "id"
.class[attrib] → Elements with class "class" and an attribute named "attrib"
.class[attrib="value"] → Elements with class "class" and an attribute called "attrib" with value **equal to** "value"
.class[attrib^="value"] → Elements with class "class" and whose attribute "attrib"value **starts** with "value"
.class[attrib*="value"] → Elements with class "class" and whose attribute "attrib"value **contains** "value"
.class[attrib$="value"] → Elements with class "class" and whose attribute "attrib"value **ends** with "value"

Example using querySelector() and querySelectorAll():

File: example1.html

```html
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <div id="div1">
      <p>
        <a class="normalLink" href="hello.html" title="hello world">Hello World</a>
        <a class="normalLink" href="bye.html" title="bye world">Bye World</a>
        <a class="specialLink" href="helloagain.html" title="hello again">Hello Again World</a>
      </p>
    </div>
    <script src="./example1.js"></script>
  </body>
</html>
```

File: example1.js

```javascript
console.log(document.querySelector("#div1 a").title); // Prints "hello world"
console.log(document.querySelector("#div1 > a").title); // ERROR: There's no immediate child inside #div1
which is a link <a>
console.log(document.querySelector("#div1 > p > a").title); // Prints "hello world"
console.log(document.querySelector(".normalLink[title^='bye']").title); // Prints "bye world"
console.log(document.querySelector(".normalLink[title^='bye'] + a").title); // Prints "hello again"

let elems = document.querySelectorAll(".normalLink");
elems.forEach(function(elem) { // Prints "hello world" and "bye world"
  console.log(elem.title);
});

let elems2 = document.querySelectorAll("a[title^='hello']");
elems2.forEach(function(elem) { // Prints "hello world" and "hello again"
  console.log(elem.title);
});

let elems2 = document.querySelectorAll("a[title='hello world'] ~ a");
elems2.forEach(function(elem) { // Prints "bye world" and "hello again"
  console.log(elem.title);
});
```

Changing the DOM

- **document.createElement("tag")** → Creates an HTML element node object. It won't be in the DOM yet, until we append it to an element which is in the DOM.

- **document.createTextNode("text")** → Creates a text node which we can append inside an element. We can do the same with element.innerText = "text".

- **element.appendChild(childElement)** → Adds a **new** child element at the last position of the current parent element.

- **element.insertBefore(newChildElement, childElem)** → Inserts a new child element before the child element passed as second parameter.

- **element.removeChild(childElement)** → Removes a child element from the current parent node.

- **element.replaceChild(newChildElem, oldChildElem)** → Replaces a child node with a new node.

Example of DOM manipulation using the same previous example's HTML:

```
let ul = document.getElementsByTagName("ul")[0];
let li3 = ul.children[2];
let newLi3 = document.createElement("li");
newLi3.textContent = "Now I'm the third element";

ul.insertBefore(newLi3, li3); // Now li3 is the fouth element of the list
li3.textContent = "I'm the fourth element...";
```

The resulting HTML will be:

```
<ul>
    <li id="firstListElement">Element 1</li>
    <li>Element 2</li>
    <li>Now I'm the third element</li>
    <li>I'm the fourth element...</li>
</ul>
```

Attributes

Inside HTML elements there are attribute nodes. Each node represents an HTML attribute like name, id, href, src, … Each attribute has a **name** and **value** property which can be read or modified (value only).

- **element.attributes** → Returns the array of the element's attributes.

- **element.className** → Used to access (read, change) the **class** attribute. We can use also **element.id, element.title** and **element.style**.

- **element.hasAttribute("attrName")** → Returns true if the element has the attribute specified.

- **element.getAttribute("attrName")** → Returns the attribute value.

- **element.setAttribute("attrName", "newValue")** → Changes attribute's value.

File: example1.html

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <p><a id="toGoogle" href="https://google.es" class="normalLink">Google</a></p>
    <script src="./example1.js"></script>
  </body>
</html>
```

File: example1.js

```
let link = document.getElementById("toGoogle");
link.className = "specialLink"; // == link.setAttribute("class", "specialLink");
link.setAttribute("href", "https://twitter.com");
link.textContent = "Twitter";
if(!link.hasAttribute("title")) {
    link.title = "Now I go to Twitter!";
}

/* <a id="toGoogle" href="https://twitter.com" class="specialLink"  title="Now I go to Twitter!">Twitter</a> */
console.log(link);
```

The style attribute

The style attribute has access to all CSS properties that can be set on an element. This attribute has a property for each corresponding CSS property, with the difference that they are in camel case format while CSS properties are in snake case format. Example: **background-color** (CSS) → **element.style.backgroundColor**. The value that can be set to each property is a string containing a valid CSS value.

File: example1.html

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <div id="normalDiv">I'm a normal div</div>
    <script src="./example1.js"></script>
  </body>
</html>
```

File: example1.js

```
let div = document.getElementById("normalDiv");
div.style.boxSizing = "border-box";
div.style.maxWidth = "200px";
div.style.padding = "50px";
div.style.color = "white";
div.style.backgroundColor = "red";
```

I'm a normal
div

Events

When a user interacts with a web application, produces events (keyboard, mouse,…) that our code should handle. There are many events that can be captured and processed (complete list here). We'll be reviewing some common events.

Page events

These events are produced by the document, usually affecting the **body** element.

- **load** → This events fires when the document finishes loading. It's used to perform actions that require the DOM to be completely loaded.

- **unload** → Occurs when the document is destroyed, for example after closing the tab where the page is loaded.

- **beforeunload** → Happens just before unloading / closing the page. By default a confirmation message appears asking the user if he/she really wants to leave the page, but other actions can also be executed.

- **resize** → This event is fired when the size of the document is changed (usually because the window is resized).

Keyboard events

- **keydown** → The user presses a key. If the key stays pressed for a while, it will generate this event repeatedly.

- **keyup** → Fires when the user releases the pressed key.

- **keypress** → More or less the same as **keydown**.

Mouse events

- **click** → This event happens when the user clicks on an element (presses and releases button → mousedown + mouseup). Also usually fired when a touch event is received.

- **dblclick** → Fired when a double click is performed on the same element.

- **mousedown** → This event occurs when the user presses a mouse button.

- **mouseup** → This event occurs when the user releases a mouse button.

- **mouseenter** → Fires when the mouse pointer enters (moves onto) an element.

- **mouseleave** → Fires when the mouse pointer leaves (moves out of) an element.

- **mousemove** → This event is repeatedly fired when the mouse pointer moves while it's inside an element.

Touch events

- **touchstart** → Fires when a finger is placed on the touch screen.

- **touchend** → Fires when a finger is removed from the touch screen.

- **touchmove** → This event is fired when a finger is dragged across the screen.

- **touchcancel** → This event occurs when the touch is interrupted.

Form events

- **focus** → This event is fired when an element (not only a form element) gets focus (is the selected or active element).

- **blur** → Fires when an element loses focus.

- **change** → Fires when the content, selection or checked state of an element changes (only <input>, <select>, and <textarea>)

- **input** → This event occurs when the value of an <input> or <textarea> element is changed.

- **select** → This event occurs when the user selects some text in an <input> or <textarea> element.

- **submit** → Fires when a form is submitted (the submission can be canceled).

Event handling

There's more than one way to assign some code or a function to handle an event. We'll see both ways (to help understand legacy code made by others), but the recommended (and valid in this course) way is to use event listeners.

Legacy event handling (not recommended)

First of all, we can put JavaScript code (or a function call) inside special HTML attributes in the element. Those attributes are named like the event with the prefix 'on'. Let's see an example of a click event.

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <div id="div1">
      <p>
        <input type="text" onclick="alert('You have clicked me!')" />
      </p>
```

```
    </div>
    <script src="./example1.js"></script>
  </body>
</html>
```

If we call a function, we can pass arguments to it (if the function requires them). The keywords **this** and **event** can be used to pass the element object and event object respectively to the function.

File: example1.html

```html
<input type="text" id="input1" onclick="inputClick(this, event)" />
```

File: example1.js

```javascript
function inputClick(element, event) {
    // Will show "A click event has been detected on input1"
    alert("A " + event.type + " event has been detected on " + element.id);
}
```

We can also add this event handlers from JavaScript code by accessing the same properties (onclick, onfocus, …), or set them to null if we want to stop handling that event.

```javascript
let input = document.getElementById("input1");
input.onclick = function(event) {
    // Inside this function, 'this' keyword refers to the element
    alert("A " + event.type + " event has been detected on " + this.id);
}
```

Event listeners (recommended)

The method to handle events explained above have some disadvantages, for example, not being able to assign more than one function to an event at the same time.

To add an event listener we call the **addEventListener** method on the element. This method receives at least 2 arguments. The event's type (string) and the function that will handle it (anonymous function or the name of an existing function).

```javascript
let input = document.getElementById("input1");
input.addEventListener('click',  event => {
    alert("A " + event.type + " event has been detected on " + this.id);
});
```

We can add as many event listeners as we want to an element. However, if want to remove an event listener, we should assign the functions first to variables, because we'll have to use those references to remove an event listener.

```javascript
let inputClick = event => {
    console.log("A " + event.type + " event has been detected on " + this.id);
};
let inputClick2 = event => {
    console.log("I'm another event handler for the click event!");
};

let input = document.getElementById("input1");
// Adding many event listeners
```

```
// Will print "A click event has been detected on input1" and "I'm another event handler for the click event!"
input.addEventListener('click', inputClick);
input.addEventListener('click', inputClick2);

// This is how we remove event listeners
input.removeEventListener('click', inputClick);
input.removeEventListener('click', inputClick2);
```

Event object

The event object is created by JavaScript and passed to the event handler functions as an argument. This object has some general properties (independent of the event type) and other properties specific to the type (for example a mouse event will have mouse pointer coordinates, or a key event will have the involved key, …).

These are some general properties that all event objects have:

- **target** → The element that triggered the event (was clicked, etc...).

- **type** → Name of the event: 'click', 'keypress', …

- **cancelable** → Returns true or false. If the event is cancelable means that we can call **event.preventDefault()** to cancel its default action (form submission, click on a link, etc…).

- **bubbles** → Returns true or false depending if this event is a bubbling event (we'll see that soon).

- **preventDefault()** → This method prevents the event's default behavior (load a new page when a link is clicked, or a form is submitted, etc…)

- **stopPropagation()** → Prevents this event from bubbling and being captured by other event handlers.

- **stopImmediatePropagation()** → If the event has more than 1 listener attached to it, calling this method prevents other listeners from executing.

Depending on the type of the event, the event object will have different properties (MouseEvent, KeyboardEvent, FocusEvent, AnimationEvent, …).

MouseEvent

- **button** → Returns which mouse button was pressed (0: left button, 1: wheel button, 2: right button).

- **clientX, clientY** → Mouse coordinates relative to the element where the event was fired.

- **pageX, pageY** → Mouse coordinates relative to the HTML document. If the page is scrolled, the scroll will be added (clientX and clientY don't add).

- **screenX, screenY** → Mouse coordinates relative to the screen.

- **detail** → Number indicating how many times the mouse button was clicked (single, double or triple click).

KeyboardEvent

- **key** → Returns the key (name) that was pressed.

- **keyCode** → Returns the Unicode character code in a **keypress** event, or the Unicode key code in a **keyup** or **keydown** event.

- **altKey, ctrlKey, shiftKey, metaKey** → Returns if the "**alt**", "**control**", "**shift**" or "**meta**" key was pressed during the event (useful for key combinations like ctrl+c). The **MouseEvent** object also has these properties.

Event bubbling

Most of the time, there are elements on a web page that overlap other elements (are contained inside). For example, if we click on a paragraph that's inside a DIV element, it's the event processed by the paragraph, by the DIV or both?. Who processes it first?.

For example, let's wee what happens with these two elements (a <div> inside another <div> when we click inside each of them).
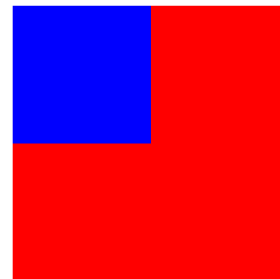
File: example1.html

```
<div id="div1" style="background-color: red; width: 200px; height: 200px;">
    <div id="div2" style="background-color: blue; width: 100px; height: 100px;"></div>
</div>
```

File: example1.js

```
let divClick = function(event) {
   console.log("you have clicked " + this.id);
};

let div1 = document.getElementById("div1");
let div2 = document.getElementById("div2");

div1.addEventListener('click', divClick);
div2.addEventListener('click', divClick);
```

If we click on the red <div id="div1"> element, it will print only "you have clicked div1". However, if we click on the blue <div> it will print both messages ("div2" first). In conclusion, by default the element which is **at the front** (usually child element) receives the event **first** and then will **bubble** it to the ancestor elements.

The event goes through 2 phases. The capture phase where the parent captures the event and goes down to the child (target) element, and the bubble phase, where the event goes from the child up to the parent element. By default, the event is processed in the bubble phase by the parent (after), but we can change that (capture phase) adding a third parameter to the addEventListener method set to **true**.

Let's see another example:

File: example1.html

```html
<div id="div1" style="background-color: green; width: 150px; height: 150px;">
    <div id="div2" style="background-color: red; width: 100px; height: 100px;">
        <div id="div3" style="background-color: blue; width: 50px; height: 50px;"></div>
    </div>
 </div>
```

File: example1.js

```javascript
let divClick = function(event) {
    // eventPhase: 1 -> capture, 2 -> target (clicked), 3 -> bubble
    console.log("you have clicked " + this.id + ". Phase: " + event.eventPhase);
};

let div1 = document.getElementById("div1");
let div2 = document.getElementById("div2");
let div3 = document.getElementById("div3");

div1.addEventListener('click', divClick);
div2.addEventListener('click', divClick);
div3.addEventListener('click', divClick);
```
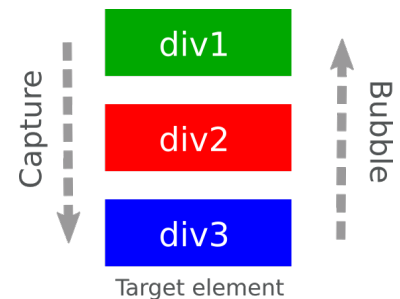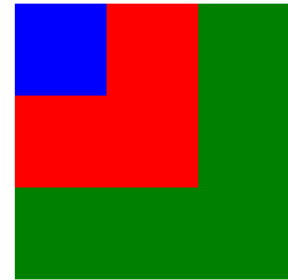
By default, when we click on the blue <div> (div3), it will print:

you have clicked div3. Phase: 2 → Target element
you have clicked div2. Phase: 3 → Bubbling
you have clicked div1. Phase: 3 → Bubbling

If we set a third argument to **true**, it would print:

you have clicked div1. Phase: 1 → Propagation
you have clicked div2. Phase: 1 → Propagation
you have clicked div3. Phase: 2 → Target element

If we call the **stopPropagation** method on the event, it won't go down (if it's in the capture phase) or up (bubbling phase).

```javascript
let divClick = function(event) {
    // eventPhase: 1 -> capture, 2 -> target (clicked), 3 -> bubble
    console.log("you have clicked " + this.id + ". Phase: " + event.eventPhase);
    event.stopPropagation();
};
```

Now, when we click the element, it will print only one message, "you have clicked div3. Phase: 2", if the third argument is not set (or set to false), or "you have clicked div1. Phase: 1" if the third argument is set to true (the parent processes the event and then prevents other child elements from receiving it).

Global objects and functions

JavaScript has some global functions and objects which can be accessed anywhere in the code. Those functions and objects provide useful functionality to work with numbers, strings, etc.

Global functions

- **parseInt(value)** → Transforms any value into an integer value. Returns the integer value or NaN if it can't be converted.

- **parseFloat(value)** → Same as parseInt but returns a float value.

- **IsNaN(value)** → Returns true if the value is NaN.

- **isFinite(value)** → Returns true if the value is a finite number or false if is infinite.

- **Number(value)** → Transforms a value into a number (or NaN).

- The above functions have been put inside the global Number object in ES2015. Number.parseInt(value), Number.parseFloat(value), Number.isNaN(value), Number.isFinite(value), Number.isInteger(value).

- **String(value)** → Converts a value into a string.

- **encodeURI(string), decodeURI(string)** → Transforms a string into a URL encoded string encoding special characters with the exception of: , / ? : @ & = + $ #. Use decodeURI to transform the string back.

  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"

  - Encoded: "http://domain.com?val=1%202%203&val2=r+y%256"

- **encodeURIComponent(string), decodeURIComponent(string)** → These functions also encode and decode the special characters that encodeURI doesn't.

  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"

  - Enc: "http%3A%2F%2Fdomain.com%3Fval%3D1%202%203%26val2%3Dr%2By%256"

Math object

The Math object provides some useful mathematical constants and methods.

- **Constants** → E (Euler's number), PI, LN2 (natural logarithm of 2), LN10, LOG2E (base-2 logarithm of E), LOG10E, SQRT1_2 (square root of ½), SQRT2.

- **round(x)** → Rounds x to the nearest integer.

- **floor(x)** → Rounds x downwards (5.99 → 5. Removes decimal part)

- **ceil(x)** → Rounds x upwards (5.01 → 6)

- **min(x1,x2,...)** → Returns the lowest number of the arguments passed.

- **max(x1,x2,...)** → Returns the highest number of the arguments passed.

- **pow(x, y)** → Returns x^y (x to the power of y).

- **abs(x)** → Returns the absolute value of x.

- **random()** → Returns a random decimal number between 0 and 1(not included).

- **cos(x)** → Returns the cosine value of x (is in radians).

- **sin(x)** → Returns the sine value of x.

- **tan(x)** → Returns the tangent of x.

- **sqrt(x)** → Returns the square root of x

```javascript
console.log("Square root of 9: " + Math.sqrt(9));
console.log("PI value is: " + Math.PI);
console.log(Math.round(4.546342));
// Random number between 1 and 10
console.log(Math.floor(Math.random() * 10) + 1);
```

In ES2015, the Math object has been extended with many new methods:

- **Hyperbolic methods** → cosh(), sinh(), acosh(), asinh(), tanh(), atanh(), hypot().

- **Arithmetic methods** → **cbrt()**: cube root ($\sqrt[3]{n}$), **expm1()**: equal to exp(x) – 1, **log2()**: log base 2, **log10()**: log base 10, **log1p()**: equal to log(x+1),…

- **Miscellaneous** → **sign()**: a number's sign (0, -0, 1, -1, NaN), **trunc()**: removes the decimal part of a number (doesn't round),...

Dates and Regular Expressions

Dates

In JavaScript we have the Date class to encapsulate information about dates. It will also store the local timezone.

```javascript
let date = new Date(); // This Date object stores actual time
console.log(typeof date); // Prints object
console.log(date instanceof Date); // Prints true
console.log(date); // Prints (at the moment of executing this) Fri Jun 24 2016 12:27:32 GMT+0200 (CEST)
```

We can pass to the constructor a number with the milliseconds since January 1st 1970 00:00:00 GMT (called **Epoch** or **UNIX time**). If we pass more than one number (only the first and second are mandatory), the order should be: 1st → year, 2nd → month (0..11), 3rd → day, 4th → hour, 5th → minute, 6th → second. The third option would be passing a string containing a date format.

```javascript
let date  = new Date(1363754739620); // New date 20/03/2013 05:45:39 (milliseconds since Epoch)
let date2 = new Date(2015, 5, 17, 12, 30, 50); // New date 17/06/2015 12:30:50 (month starts at 0 ->
January, ... 11 -> December)
let date3 = new Date("2015-03-25"); // Long string format without time YYYY-MM-DD (00:00:00)
let date4 = new Date("2015-03-25T12:00:00"); // Long string format with time
let date5 = new Date("03/25/2015"); // Short format MM/DD/YYYY
let date6 = new Date("25 Mar 2015");  // Short format with month name (March would also be valid). Order
doesn't matter here
let date7 = new Date("Wed Mar 25 2015 09:56:24 GMT+0100 (CET)"); // Full format with timezone. If omitted
browser's timezone is assumed
```

If, instead of a Date object, we'd like to get the number of milliseconds since Epoch directly (number type), we have **Date.parse**(string) and **Date.UTC**(year, month, day, hour, minute, second) methods. We can call also **Date.now()**, that will return the current date and time in milliseconds.

```javascript
let nowMs = Date.now(); // This moment in ms
let dateMs = Date.parse("25 Mar 2015"); // 25 March 2015 in ms
let dateMs2 = Date.UTC(2015, 2, 25); // 25 March 2015 in ms
```

The Date class has **setters** and **getters** for **fullYear**, **month** (0-11), **date** (day), **hours**, **minutes**, **seconds**, and **milliseconds**. If we set a negative value, for example, month = -1, it represents the last month from the previous year.

```javascript
// Create a date object the represents 2 hours ago
let twoHoursAgo = new Date(Date.now() - (1000*60*60*2)); // - 2 hours in ms
// Doing the same using the Date object set method
let now = new Date();
now.setHours(now.getHours() - 2);
```

When we want to print the date, we have methods that give us the stored date in different formats.

```javascript
let now = new Date();

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

```
console.log(now.toISOString()); // Prints 2016-06-26T18:00:31.246Z
console.log(now.toUTCString()); // Prints Sun, 26 Jun 2016 18:02:48 GMT
console.log(now.toDateString()); // Prints Sun Jun 26 2016
console.log(now.toLocaleDateString()); // Prints 26/6/2016
console.log(now.toTimeString()); // Prints 20:00:31 GMT+0200 (CEST)
console.log(now.toLocaleTimeString()); // Prints 20:00:31
```

Regular Expressions

Regular expressions let us search for patterns in strings, in order to find a substring that matches that pattern, or perform substitutions for example. In JavaScript, we can create a regular expression by instantiating the class RegExp or using an special format putting the regular expression string between two slash '/' characters.

Here we'll review only the basics about regular expressions. You can learn more about them here. Also, there are web sites like this one that let you test your regular expression against any text.

A regular expression also can have one or more options or modifiers, such as '**g**' → Global search (if not present only matches the first result), '**i**' → Case-insensitive ('A' and 'a' will be equal characters), or '**m**' → Multiline search (if omitted, every line will be independent). In Javascript you create a regular expression object with modifiers in these two ways:

```
let reg  = new RegExp("^[0-9]{2}","gi");
let reg2 = /^[0-9]{2}/gi;
console.log(reg2 instanceof RegExp); // Prints true
```

Those two forms are equivalent as you can see, so choose whichever you prefer.

Regular expressions basics

The most basic form of a regular expression is to include only alphanumeric (or any others which are not special) characters. This expression will search for a piece of a string that contains exactly these characters in the same order.

```
let str = "Hello, I'm using regular expressions";
let reg = /reg/;
console.log(reg.test(str)); // Prints true
```

In this case, "reg" is found in "Hello, I'm using **reg**ular expressions", so the test method returns true (we'll learn more about regular expression's methods soon).

Brackets

Putting characters (or a range of characters) inside brackets will match against **one** character inside those brackets.

[abc] → Any character that is 'a', 'b', or 'c'

[a-z] → Any character between a and z (lowercase letters)

[0-9] → Any character between 0 and 9 (numerical character)

[^ab1-9] → Any character **except** (^) 'a', 'b' or numerical.

Optionality (pipes → |)

(exp1|exp2) → The string must match the first **or** the second expression. You can add as many pipes as you want, and also many groups of choices in a bigger expression.

Meta-characters

. (dot) → Any single character (except newline)

\w → Word or alphanumeric character. \W → Non-word character.

\d → Digit character. \D → Non-digit character.

\s → Whitespace character. \S → Non-whitespace character.

\b → Word delimiter. Matches the beginning or the end of a word (not a character). For example /\bcase\b/, matches "case" but not "uppercase" or "casein".

\n → New line character.

\t → Tab character.

Quantifiers

+ → The precedent character (or expression inside parentheses) repeats 1 or more times.

* → Zero or more occurrences. Contrary to +, it doesn't have to appear.

? → Zero or one occurrences. We could say it's optional.

{N} → Must appear N times.

{N,M} → From N to M times.

{N,} → At least N times.

^ → Matches the beginning of a string (not any character).

$ → Matches the end of a string.

You can find more about regular expressions in JavaScript here.

Examples

/^[0-9]{8}[a-z]$/i → Matches a string containing a DNI. This expression matches

a string that starts with eight numbers, followed by a letter ('i' modifier makes it case-insensitive, so it doesn't matter if it's lowercase or uppercase), and then ends.

/^\d{2}\/\d{2}\/(\d{2}|\d{4})$/ → Matches a string containing a date in DD/MM/YY or DD/MM/YYYY format. Note that \d is equivalent to [0-9], and the year can't have three numbers (two or four).

\b[aeiou]\w*\b/i → Matches any word in a string that starts with a vowel, followed by zero or more alphanumeric characters (numbers, letters or '_').

Regular expressions methods in JavaScript

With a regular expression object, we can use two methods for testing if a string matches that expression. Whose methods are **test** and **exec**.

The **test()** method receives a string an tries to find a coincidence in the string for the regular expression. If the global '**g**' modifier is set, every time the method is executed will continue from the last position it found a match, so we can know how many matches are in that string. Note also the difference with ignore-case '**i**' modifier.

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Prints true
console.log(reg.test(str)); // Prints true
console.log(reg.test(str)); // Prints false, there are only two matches

let reg2 = /am/gi; // "Am" will match now
console.log(reg.test(str)); // Prints true
console.log(reg.test(str)); // Prints true
console.log(reg.test(str)); // Prints true. Now there are three matches.
```

If we want to get more details about matches, we could use exec() method. This method returns an object with details about the piece of the string that matches, index where the match starts, and also the entire string.

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Prints ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Prints ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Prints ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Prints null. No more matches...
```

A better way to

```
let str = "I am amazed in America";
let reg = /am/gi;
let match;
while(match = reg.exec(str)) {
    console.log("Match found!: " + match[0] + ", in position: " + match.index);
}
/* This will print:
 * Match found!: am, in position: 2
 * Match found!: am, in position: 5
 * Match found!: Am, in position: 15 */
```

Similarly, there are methods we can use from a string that accept regular

expressions as parameters. Those methods are **match** (similar to exec) and **replace**.

The method match returns an array with all the the matches found in the string if the global modifier is set. If it's not set, it will return the same as **exec()**.

```
let str = "I am amazed in America";
console.log(str.match(/am/gi)); // Prints ["am", "am", "Am"]
```

The method replace returns a string with the parts that match the regular string replaced with another string (if the global modifier is not set, only the first is replaced). We can also send an anonymous function that processes every match in the string and returns the replacement.

```
let str = "I am amazed in America";
console.log(str.replace(/am/gi, "xx")); // Prints "I xx xxazed in xxerica"

console.log(str.replace(/am/gi, function(match) {
    return "-" + match.toUpperCase() + "-";
})); // Prints "I -AM- -AM-azed in -AM-erica"
```