****

**Javascript Introduction**

**- Adding interactivity to HTML documents –**

**edX – W3CX(Microsoft)**

1. **Module 2 Outline**

.

### **What you will learn in Module 2**

* Conditional statements and logical operators: if...then...else, etc.
* Loops: iterating on collections, doing things repeatedly
* Events: detect keys, mouse clicks or moves, window resize events, etc.
* DOM: interacting with the content of the HTML page (selecting elements, modifying them, adding and removing new ones)
* HTML5 canvas intro: learn how to draw and animate simple shapes in a HTML5 canvas

1. **Conditional statements, loops and logical operators**
   1. **Boolean values and logical operators**

**[ -- Live coding video: boolean value, if...else statement and comparison operators -- ]**

### **Boolean values and logical operators**

#### Introduction

Before talking about how your JavaScript program can make decisions, such as "if this condition is fulfilled then I'll do this, otherwise I'll do that...", we need to define a few more concepts.

We will start with "boolean values" and "logical operators".

#### Boolean values

The **boolean** type represents a logical entity having two values: true and false.

Use of the keywords true and false:

1. var b = true;
3. var b = false;

A boolean variable should not be enclosed in quotation marks, otherwise it becomes a string variable:

1. var b = 'true'; // b is not a boolean but a string

#### Undefined and null values

**Undefined**

undefined is returned when a variable has not been assigned:

1. var foo;
2. > foo
3. undefined
5. > typeof foo;
6. 'undefined'
8. >if (foo === undefined) {
9. console.log('The variable foo has no value and is undefined');
10. }
11. 'The variable foo has no value and is undefined'

The above example shows how we can test whether a variable has a value (line 8 uses a conditional statement).

The keyword "**undefined**" is part of the JavaScript language, so you can assign the undefined value to a variable:

1. > var foo = undefined; // equivalent to var foo; without giving any value
2. undefined
4. > foo;
5. undefined

var foo; and var foo = undefined; are equivalent but we recommend that you use the first version to declare the variable (it is shorter, and that reduces the code).

If you try to access a variable that has not been declared before, a ReferenceError will be raised. But the typeof operator will return "undefined":

1. > bar;
2. ReferenceError
4. > typeof bar;
5. 'undefined'

#### Logical operators

The logical operators are as follows:

* **&&** (AND)    
  usage example : **if ((x  > 0) && (x < 10)) {**  
  **console.log('x is strictly positive and less than 10');**  
  **}**
* **||** (OR)  
  usage  example : **if ((x  > 0) || (x == -5)) {**  
  **console.log('x is positive or equal to -5');**  
  **}**
* **!** (NOT)  
  usage example : **if (!(x  > 0)) {**  
  **console.log('x is not positive (x is less or equal to 0');**  
  **}**
* **&&**, **||** operators are binary, **!** is unary.

1. var b = !true;
2. b; //false
4. var b = !!true;
5. b; //true
7. var b = "one";
8. !b;   false // implicit conversion of "one" to a boolean value
10. var b = "one"; // implicit conversion of "one" to a boolean value
11. !!b; //true

 In an expression with logical operators, as shown in lines 8 and 11 of the previous example, non-boolean values are implicitly converted to boolean.

#### Lazy evaluation or short-circuit evaluation:

Logical expressions are evaluated from left to right. JavaScript uses a mechanism known as "short-circuit evaluation" to prevent the second, third, and nth conditions from being tested in certain cases:

* false && something (an expression)  is always false, and the part to the right of && operator is not tested.
* true || something (an expression) is evaluated to true, and the part to the right of the || operator is not tested.

Examples:

1. var b = 5;
2. var c = 6;
4. if ((b === 5) || (b === 6))  { //the second part is never tested
5. console.log('b is equal to 5 or equal to 6');
6. }
8. if ((b === 5) && (c === 6)) {  // second part is evaluated
9. console.log('b  is equal to 5 and c is equal to 6');
10. }
12. if ((b === 15) && (c === 6)) {  // second part is never evaluated
13. console.log('b  is equal to 5 and c is equal to 6');
14. } else {
15. console.log('b not equal to 15 or c not equal to 6');
16. }

#### Implicit conversions of non boolean values in expressions

Used with logical operators or within statements, **non-boolean values are implicitly converted to booleans**.

**All the following values are evaluated as false :**

* **false**
* **undefined**
* **null**
* **0**
* **NaN**
* **the empty string ''**

**Everything else is evaluated as true!**

1. var boo = 'hello' && 'world'; // boo is equal to 'world' that is 'true'.

In the above example, 'hello' && 'world' is evaluated as true but will return a value! Indeed, boo will equal 'world' because 'hello' is a string value that is evaluated as true. And 'world' is in also true as it's not one of the value cited in the previous paragraph. If we do : if (boo) then.... we will enter the if statement.

The rule is that both && and || result in the value of (exactly) one of their operands:

* A && B returns the value A if A can be coerced into false; otherwise, it returns B.
* A || B returns the value A if A can be coerced into true; otherwise, it returns B.

External resource : More détails are available in: [The && and || Operators in JavaScript](https://mariusschulz.com/blog/the-and-and-or-operators-in-javascript)

**TO SUM UP: it works "normally" if you just think true/false, but the real value affected is not true false, it's one of the operands, that can be seen as true/false.**

1. **var boo2 = (0/0) || 43.2 ;**

 boo2 equals 43.2 because the expression 0/0 equals NaN, which is evaluated as false.

**Question:**

What is the value of the variable myNumber after the execution of this code?

1. var myNumber = !1;
3. if(myNumber == null){
4. myNumber = 3;
5. }
7. myNumber = myNumber || 2;

**Explanation**: after the first line, myNumber equals false. In the if statement, at line 3, false does not equal null. Therefore, the value 3 is not assigned to the var myNumber. In the last line, myNumber is evaluated as false, then the value 2 is given to the variable myNumber.

#### Comparison Operators

* Equal **==**
* Not equal **!=**
* Greater than **>**
* Greater than or equal **>=**
* Less than **<**
* Less than or equal to **<=**
* Strict equal **===**
* Strict not equal **!==**

#### What is the difference between ****==**** and === in JavaScript?

##### **Equal (==)**

Returns true if the operands are strictly equal **with type conversion**.

##### **Strict equal (===)**

Returns true if the operands are strictly equal **with no type conversion**.

The triple-equals operator never does type coercion. It returns true if both operands reference the same object, or in the case of value types, have the same value.

Some examples :

1. 1 == 1 ;
2. //true
4. 1 == 2 ;
5. //false
7. /\* Here, the interpreter will try to convert the string ‘1’
8. into a number before doing the comparison \*/
10. 1 == '1';
11. //true :

14. //with strict equal, no conversion:
16. 1 === 1;
17. //true
18. 1 === '1';
19. //false

**Depending on the context, generally strict equal (or strict not equal) is preferred.**

**Best practice for beginners: always use === or !== for comparisons.**

Here's an interesting article about this question:

* "[Why you should use strict equal](http://www.impressivewebs.com/why-use-triple-equals-javascipt/)" (Impressive Webs, March 1st, 2012)

#### Specific case of NaN

As we have already seen, JavaScript has some special values. One of them is NaN: “Not-a-Number”.

NaN has this special property :

1. NaN == NaN;
2. // false
4. NaN === NaN;
5. // false

**Nan is equal to nothing - not even to itself!**  But you do have a function to check the NaN value:  isNaN(expr)

* isNaN: returns true if the argument coerces to NaN, and otherwise returns false.

1. isNaN(NaN);
2. // true
4. isNaN(0/0);
5. // true
7. isNaN(12);
8. // false
10. isNaN('foo');
11. // true

"A reliable way for ECMAScript code to test if a value X is a NaN, is an expression of the form X !== X. The result will be true if, and only if, X is a NaN. " (see the [isNan documentation](http://www.ecma-international.org/ecma-262/5.1/#sec-15.1.2.4)).

**A complete example with isNaN:**

1. var num =0/0;
3. if(isNaN(num)){
4. num = 0;
5. }
6. //shortened version with the conditional operator
7. var num = isNaN(num) ? 0 : num
9. //version with logical operator (implicit conversion)
10. var num = num || 0;
12. /\*
13. <=> num = NaN || 0
15. <=> num = false || 0
16. \*/
18. num;
19. //returns 0 in this three cases

Of course 0/0 rarely happens, but there are other cases where NaN can appear, for example:

* parseInt('foo');  returns NaN   //parseInt tries to convert a String to a Number
* Math.sqrt(-1); return NaN

**Knowledge check 2.2.1 (not graded)**

var a = 5;

if ((a === 5) || (a === 6)) {

console.log('a is equal to 5 or equal to 6');

}

**When will the expression (a === 6) be evaluated?**

1. never
2. before (a === 5)
3. after (a === 5)
   1. **Conditional statements: if..then…else, switch**

**[-- live coding video: switch statement --]**

### **Conditional statements**

#### What are statements?

JavaScript source code is a set of statements. There are a couple of different statement types. We have already seen one of them, the variable statement:

1. var myVar = 'hello ' + 'world';

We've also seen the expression statement:

1. 3 + 4;
3. // more often like this
4. var x = (3 + 4);
5. var y = (5 + x);

A statement closes with a semicolon, but we will see later that missing semicolons are automatically inserted (for readability reasons, we highly recommend systematically adding a semicolon at the end of all statements).

Statements are generally executed sequentially from top to bottom of the script. However, this flow can be modified by statements such as conditional statements and iteration statements.

#### The block statement

The block statement is a simple statement which allows us to group a set of statements wrapped in curly braces.

Block statement:

1. {
2. var i = 0;
3. var result = false;
4. console.log('i = ' + i);
5. }

The block statement is used by other statements such as the if-statement or for-statement. We will see these statements below.

#### Conditional statements

All the examples for this section

|  |
| --- |
| <script>    /\*CONDITIONAL STATEMENTS\*/    /\*if statement\*/    var num = 10;    if (num == 10) {      num = 20;    }    console.log('if statement,\n num>> ' + num);    /\*if-else statement\*/    var num = 10;    if (num > 10) {      num = 20;    } else {      num = 0;    }    console.log('if-else statement,\n num>> ' + num);    /\*    Question 1    how to replace this code by an expression including one operator?    \*/    var max;    var min;    // Try to uncomment that and // see the difference!    // var min=1;    if (min) {      max = min + 10;    } else {      max = 10;    }    console.log('question1, \n max>> ' + max);    //Answer is at the end of the document    /\*switch statement\*/    //switch statement behaviour when break keyword is missing    //comment and uncomment cloudColor variable to see the different results    var gear = '';    //var cloudColor;    //var cloudColor = 'green';    var cloudColor = 'black';    switch (cloudColor) {      case 'green': gear += 'spacesuit';        break;      case 'black': gear += 'boots, ';      case 'grey': gear += 'umbrella, ';      case 'white': gear += 'jacket, ';      default: gear += 'watch';    }    console.log('switch2,\n gear >> ' + gear);    //The above example with break;    var gear = '';    //var cloudColor;    //var cloudColor = 'green';    var cloudColor = 'black';    switch (cloudColor) {      case 'green':        gear += 'spacesuit';        break;      case 'black':        gear += 'boots, ';        break;      case 'grey':        gear += 'umbrella, ';        break;      case 'white':        gear += 'jacket, ';        break;      default:        gear += 'watch';    }    console.log('swtich3,\n gear >> ' + gear);    /\*    Question 1    how to replace this code by an expression including one operator?    var max;    var min;    if (min){        max = min + 10;    } else {        max = 10;    }    \*/    //Answer    var max;    var min;    max = (min) ? min + 10 : 10;  //console.log('question 1,\n max >> ' + max);  </script> |

(Please look, edit and try whatever you want. There are parts that are commented - please remove comments and try to understand the results).

**Conditional statements are used to execute a unit of code  
only if a condition is evaluated as true.**

#### The if statement

**Syntax:**

**if** ( Expression ) Statement **else** Statement

**if** ( Expression ) Statement

**The expression may include:**

* logical operators ( **! && ||** )
* comparison operators ( **==**, **===**, **>**, **>=**, **<**, **<=** )
* any values or expressions which can be converted to boolean

##### **Example: if-statement**

1. var num = 10;
3. **if** (num === 10) **{**
4. num = 20;
5. **}**
7. // num equals 20

##### **Example: if-else statement**

1. var num = 10;
3. **if** (num > 10) **{**
4. num = 20;
5. **}** **else** **{**
6. num = 0;
7. **}**
9. // num equals 0

**Reminder:**

The following values will evaluate to false:

* **false**
* **undefined**
* **null**
* **0**
* **NaN**
* **""** (empty string)

**All other values, including all objects, evaluate to true when passed to a conditional statement.**

#### The if-then-else ternary operator

This ternary operator is a shortcut version of if...then...else.

Let's look at this code example:

1. var max;
2. var min = 2;
4. if (min < 10) {
5. max = min + 10;
6. } else {
7. max = min;
8. }

Explanation: You can replace this "if-then-else" statement with the ternary operator that uses a syntax with "?" and ":"

1. var max;
2. var min;
3. max = (min < 10)? min+10 : min;

Line 3 can be read as if (min < 10) then max = min+10, else max = min. The "then" part is after the "?" and the "else" part is after the ":" part.

This "short" version is not recommended except for very simple statements that involve a very obvious block of instructions for the "then" and the "else". Usually this syntax is much harder to read for beginners.

#### Curly braces

Should we use them in if-then-else statements? There are examples without curly braces on the Web: what does this mean?

Here are two versions of the same code.

**Version 1: no curly braces**

1. **if** (a > 2)
2. result = 'a is bigger than 2';
3. **else**
4. result = 'a is not bigger than 2';

**Version 2: with curly braces for delimiting the "then" and "else" blocks**

1. **if** (a > 2) **{**
2. result = 'a is bigger than 2';
3. **} else {**
4. result = 'a is not bigger than 2';
5. **}**

Version 1 and version 2 are equivalent. Indeed, version 1 is correct: you can omit curly braces if the "then" or "else" blocks are made of only one statement (one line of code).

But version 2 is cleaner and more readable, and, in particular, it is much better for maintainability (because you can add a statement just by pressing the enter key. And you can add some extra lines of code without worrying about adding curly braces because you broke the "1 line statement rule").

**So it is strongly recommended that you always use if-statements  
enclosed in curly braces.**

Of course, one-line if-statements like this :

1. if (true) doSomething();

...are really fast to write, but if you want to add a second statement later it will become more time consuming.

**Conclusion: always use curly braces!**

#### The switch statement

In order to avoid having a series of ifs and elses, it is possible to use a **switch** statement.

The syntax of the switch statement is:

1. **switch** (expression) **{**
2. **case** value1:
3. statement
4. **break**;       // break can be omitted in that case
5. // the second test case will be executed
6. // most of the time we add a break; at the end
7. // of a "case"
9. **case** value2:
10. statement
11. **break**;
13. **case** value3:
14. statement
15. **break**;
17. **default**:         // if no case tested true
18. statement
19. **break**;
20. }

If the value of an expression equals one of the cases (the equality operator evaluated is**===**), all the statements next to this case block are executed sequentially until the keyword **break** is reached.

**Example 1: a common switch/case/default example.**

1. var gear = '';
3. **switch** (cloudColor) **{**
4. **case** 'green':
5. gear = 'spacesuit';
6. break;
8. **case** 'black':
9. gear = 'boots';
10. **break**;
12. **case** 'grey':
13. gear = 'umbrella';
14. **break**;
16. **case** 'white':
17. gear = 'jacket';
18. **break**;
20. **default**:
21. gear = 'watch';
22. **break**; // useless if in the last case
23. **}** // end of the switch statement

In this example, if the clouds are grey, then my gear will be just an umbrella. If they are white, I'll wear only a jacket, if they are black I'll be nude with just boots (!), and if they are green I'll get a spacesuit. And if the cloud color is none of these, then I'll only wear a watch. The presence of the **break** keyword at the end of the different cases make the choices 100% exclusive. Only one case can be executed!

**Example 2: a switch without "breaks" at the end of each case.**

1. var gear = '';
3. **switch** (cloudColor) **{**
4. **case** 'green':
5. gear += 'spacesuit';
6. **break**;
8. **case** 'black':
9. gear += 'boots, ';
11. **case** 'grey':
12. gear += 'umbrella, ';
14. **case** 'white':
15. gear += 'jacket, ';
17. **default**:
18. gear += 'watch';
19. **}** // end of the switch statement

Explanation: if the clouds are black, then my gear will be 'boots, umbrella, jacket, watch'. If the clouds are green, my gear is a spacesuit (because of the break keyword, other cases will not be tested). If the cloud color is not in the listed colors, then my gear is only a watch (default case).

To finish up this section, here is a complete example: three ways to do condition statements

|  |
| --- |
| <script>    /\*CONDITIONAL STATEMENTS\*/    /\*3 examples which are equivalent\*/    //try to change foo value    var foo = 1;    //var foo=2;    //var foo=1000;    //var foo=0;    //var foo='1';    var bar1, bar2, bar3;    //example 1    if (foo === 1) {      bar1 = 'one';    }    else if (foo === 2) {      bar1 = 'two';    }    else {      bar1 = 'something';    }    //example 2    bar2 = (foo === 1) ? 'one' : (foo === 2) ? 'two' : 'something';    //example 3    switch (foo) {      case 1:        bar3 = 'one';        break;      case 2:        bar3 = 'two';        break;      default:        bar3 = 'something';    }    //now we print results :    console.log('example1,\n bar1 >> ' + bar1);    console.log('example2,\n bar2 >> ' + bar2);    console.log('example3,\n bar3 >> ' + bar3);  </script> |

**Knowledge check 2.2.2 (not graded)**

var x = 2;

var y = 5;

if(y > 0) {

if ((x > 2) && (y < 10)) {

if(x === 2) console.log("YES");

} else {

if((x > 10) && (y === 5)) console.log("NO");

}

} else {

console.log("MAYBE");

}

console.log(" THE END");

**What will be printed in the devtool console?**

1. THE END
2. NO THE END
3. YES THE END
   1. **Loop statements**

**[ -- live coding video: loop statements -- ]**

#### Loops

A loop is used to run the same block of code several times while a condition is satisfied.

If you have trouble with loops, the online tool [slowmoJS](http://toolness.github.io/slowmo-js/) can be really useful: you just have to copy and paste an example into it to run it step by step and see how your program executes loops.

#### The while statement

With a **while** statement, a block of code is executed repeatedly while the specified condition is satisfied (evaluates to true).

Syntax:

1. while ( condition ) statement

The condition is an expression, and the statement can be a block statement.

Typical example of a while statement:

1. var i = 1, j = 1;
3. **while ( i < 4 ) {**
4. j += i;
5. i += 1;
6. **}**
7. ...

The block inside the while (lines 4 and 5) will be executed three times:

* Line 1 initializes i with a value of 1.
* We enter the while statement at line 3. Is the value of i strictly less than 4?
* Yes, the variable i is equal to 1, we enter the statement inside the while.
* **Run 1:**
  + We execute line 4: j += i; (equivalent to j = j + i). As j was set to 1 at line 1, j is now equal to 2.
  + We execute line 5 and increment i by one. The variable i is now equal to 2.
  + We go back to the while at line 3. Is i < 4? Yes, we execute lines 3 and 4 again.
* **Run 2:**
  + Now at the end of line 5, j is equal to "old j value" + "new i value", so j = 2 + 2 = 4, i has been incremented and is now equal to 3.
  + We go back to the while at line 3. Is i < 4? Yes, we execute lines 3 and 4 again.
* **Run 3:**
  + Now at the end of line 5, j is equal to "old j value" + "new i value", so j = 4 + 3 = 7, i has been incremented and is now equal to 4.
  + We go back to the while at line 3. Is i < 4? No! The value of i is now 4, which is not less than 4. We continue the execution of the program at line 7 with i = 4 and j = 7.

Of course, if the condition never evaluates to false, the block will be executed infinitely until the machine crashes... a test like while (i > 0) { .....} will never stop and will eat all the CPU.

Try this example now with [slowmoJS](http://toolness.github.io/slowmo-js/?code=var%20i%20%3D%201%2C%20j%20%3D%201%3B%0A%20%0Awhile%20(%20i%20%3C%204%20)%20%7B%0A%20%20%20%20j%20%3D%20j%20%2B%20i%3B%0A%20%20%20%20i%20%3D%20i%20%2B%201%3B%20%0A%7D&filterrange=65-65) !

#### The do-while statement

The do-while statement is very similar to the while statement, but its syntax is different:

1. do statement while ( condition )

Typical example:

1. var i = 0;
3. **do {**
4. **console.log('i = ' + i);**
5. i++;
6. **} while(i < 20);**
8. console.log('Value of i after the do-while statement: ' + i);

The **do-while statement executes the content of the loop once before checking the condition of the while**, whereas a **while statement will check the condition first before executing the content**.

A do-while is used for a block of code that must be executed at least once.These situations tend to be relatively rare, thus the simple while-statement is more commonly used.

If you want to "see" the difference, [look at the "do-while" statement with slowmoJS](http://toolness.github.io/slowmo-js/?code=var%20condition%3D%20false%3B%0Avar%20foo%20%3D%200%3B%0Ado%7B%0A%20foo%2B%2B%3B%0A%7D%20while(condition%20%3D%3D%20true)%3B%0A%0Afoo%3B&filterrange=80-80) and [the "while" statement slowmoJS](http://toolness.github.io/slowmo-js/?code=var%20condition%3D%20false%3B%0Avar%20foo%20%3D%200%3B%0Awhile(condition%20%3D%3D%20true)%3B%20%7B%0Afoo%2B%2B%0A%7D%0A%0Afoo%3B&filterrange=68-68).

#### The for statement

This statement adds some things to the while and do-while statements: an initialization expression and an incrementing expression.

Its syntax is:

1. for (initialization; condition; incrementation) statement

The three expressions within the parentheses are optional. If the condition is omitted, it is replaced by true (infinite loop).

Typical example (counting from 0 to 10):

1. for (var i = 0; i <= 10; i++) {
2. console.log('i = ' + i);
3. }

We can have more than one instruction in the "initialization part", and more than one instruction in the "incrementation part". Here is another example:

1. for (**var i = 1, j = 1;** i <= 10;**i++, j+=2**) {
2. console.log('i = ' + i + ' j = ' + j);
3. }

In this example, two variables are defined and assigned within the initialization expression. Before each execution of the block statement, the condition is checked; here we need i <=10. After each execution of the block statement, the incrementation expression is executed to increment the variables i by 1 and j by 2.

Open the devtool console of your browser and copy and paste the above code, or [look at the slowmoJS execution](http://toolness.github.io/slowmo-js/?code=for%20(var%20i%20%3D%201%2C%20j%20%3D%201%3B%20i%20%3C%3D%2010%3B%20i%2B%2B%2C%20j%2B%3D2)%20%7B%0A%20%20%20%20console.log(%27i%20%3D%20%27%20%2B%20i%20%2B%20%27%20j%20%3D%20%27%20%2B%20j)%3B%0A%7D&filterrange=0-89).

#### The for-in statement

The for-in statement is used to iterate through an object (or through an array, which is also an object).

Its syntax is:

1. for ( variable in expression ) statement

Typical example:

1. var michel = {              // michel is an object
2. familyName:'Buffa',     // familyName, givenName, age
3. // are its properties
4. givenName: 'Michel',
5. age: 51
6. }
8. **for(var property in michel) {**   // the for-in will
9. // enumerate properties
10. console.log(property);      // will print "familyName",
11. // "givenName",
12. // "age"
13. console.log(michel[property]);  // michel['givenName'] same
14. // as michel.givenName
15. **}**

Before each execution of the block statement, the variable named "property" is assigned with the name of one of the properties (the keys) of the object.

We will see further examples of this statement in module 4, which is devoted to the study of JavaScript objects.

#### [ADVANCED] Other statements

##### **The continue statement**

The continue statement is used to stop the execution of a block and start the next iteration of the loop. The difference from the "break" statement is that the loop continues.

Syntax:

1. continue [label]

The label is optional.

Typical example:

1. for(var i = 1, k = 0; i < 5; i++) {
2. if (i === 3) {
3. continue;
4. }
6. k += 2\*i;
7. console.log('k += ' + (2\*i));
8. }
9. console.log('Final k value:' + k)

Copy and paste this example in your devtool console, but first, try to guess what the value of k will be!

Hint: lines 2-4 mean that line 6 will never be executed for i = 3. That means that i\*2 will only be added to k for i = 1, 2 and 4...

##### **The break statement**

The break statement is used to stop an iteration, a switch or a labelled statement.

Syntax:

1. break [label]

Typical example:

1. var tab = ['michel', 'john', 'donald', 'paul']; // johh at index = 1
3. function isNameInTheArray(name, theArray) {
4. console.log("Number of elements in the array : " + theArray.length);
5. for(var i=0; i < theArray.length; i++) {
6. console.log('comparing with element in the array at pos ' + i);
8. if(theArray[i] === name) {
9. console.log('the name ' + name +
10. ' is in the array at pos: ' + i);
11. **break;**
12. } else {
13. console.log(name + ' is not at pos ' + i);
14. }
15. }
16. }
18. // Execute the function
19. isNameInTheArray('john', tab);

Copy and paste in the devtool console. You'll see that the function that compares each element in the array passed as the second parameter with the name 'john', will stop looping after 'john' has been found at index = 1.

**Detailed explanations:**

* Line 20 executes the function
* Line 6: The for statement loops on all existing indexes in the tab, from 0 to tab.length
* Line 9: if the condition is true, we enter the block and execute lines 10-12
* The break statement at line 12 will exit from the loop, it "breaks" the loop.
* The different console.log(...) will never display the message "comparing with elements..." with indexes greater than 1: the loop exists when 'john' is found at index 1 (i equal to 1).

**Knowledge check 2.2.3 (not graded)**

for(var i = 0; i < 20; i++) {

// the loop block of instructions

// do something...

}

**How many times will the loop block of instructions be executed?**

1. 21 times
2. 20 times
3. 19 times
   1. **Discussion and projects**

Here is the discussion forum for this part of the course. Please either post your comments/observations/questions or share your creations.

See below for suggested topics of discussion and optional projects.

#### Suggested topic of discussion:

* Did you know about the === and !== operators we recommend you to use?

#### Optional projects:

* If not allergic to High School math: try to write a piece of code that solves second degree equations. You pass the a, b, c parameters of: ax2 + bx + c, and the solve function will compute delta = b2 - 4a\*c. Test the sign of delta, and if it's equal to zero, then display (in the console, or better, in the page) the roots of the equation.
* Try to write a small program that asks you to guess a number. It will choose randomly a number, and will ask you to enter a value in an input field. Then it will display "too small" or "too big", until you find the number.  
    
  Hint: use the Math.random and Math.round methods, such as in let randomNumber = Math.round(Math.random() \* 10); to get a random value between 0 and 10.  
    
  For working with input fields, look at section 1.4 from the first module, the math function plotter example used input fields. Or look at the section about DOM in this module.
* Try to make a quiz using the DOM and buttons, checkboxes, etc.
* Display a question, for example "Which actor played in Titanic?", and display two or three buttons ("Leonardo Di Caprio", "Christian Bale", "Nicolas Cage"). Then, when the user presses a button, you must check the answer and display the next question, etc.
* Use CSS with an image background for the buttons.
* Or use images with click listeners - we saw this in the section about the DOM and events.
* A bit more challenging: use checkboxes instead of a set of buttons (we've done an example close to this in the section that presented the DOM).

1. **Functions (part2): callbacks**
   1. **Functions and callbacks**

#### Two ways to declare a function

##### 1 - Standard function declaration

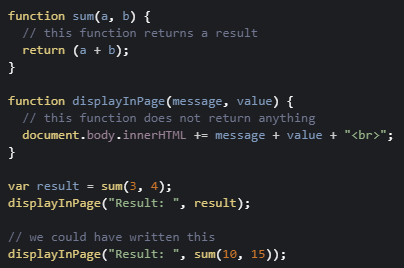
We've already seen that functions can be declared using this syntax:

1. function functionName(parameters) {
2. // code to be executed
3. }

A function declared this way can be called like this:

1. functionName(parameters);

Notice that we do not add a semicolon at the end of a function declaration. Semicolons are used to separate executable JavaScript statements, and a function declaration is not an executable statement.

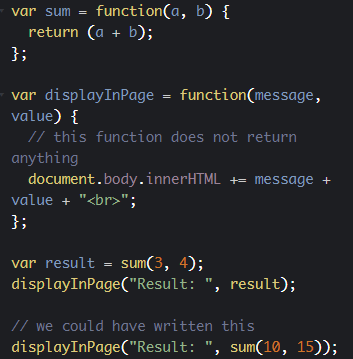
****

In the above example, the sum function returns a value, and the displayInPage function does not return anything.

##### **2 - Use a function expression**

A JavaScript function can also be defined using an expression that can be stored in a variable. Then, the variable can be used as a function:

Here is a typical example:

****

Notice how the sum and displayInPage functions have been declared. We used a variable to store the function expression, then we can call the functions using the variable name. And we added a semicolon at the end, since we executed a JavaScript instruction, giving a value to a variable.

The "function expression" is an "anonymous function", a function without a name, that represents a value that can be assigned to a variable. Then, the variable can be used to execute the function.

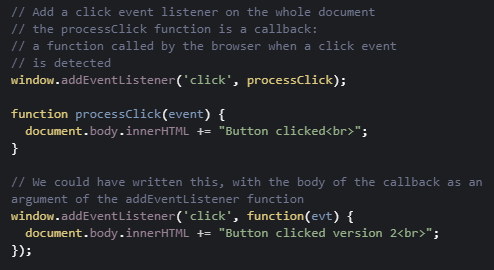
We say that functions are "first class objects" which can be manipulated like any other object/value in JavaScript.

This means that functions can also be used as parameters to other functions. In this case they are called "callbacks".

#### Callbacks

Indeed, as functions are first-class objects, we can pass a function as an argument, as a parameter to another function and later execute that passed-in function or even return it to be executed later. When we do this, we talk about callback functions in JavaScript: a function passed to another function, and executed inside the function we called.

All the examples of event listeners that you've seen used callback functions. Here is another one that registers mouse click listeners on the window object (the window objects represent the whole HTML document):

****

****

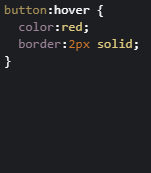
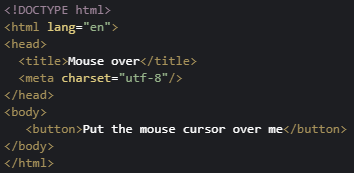
****

In this case, the processClick function is passed as a parameter to the addEventListener method/function.

Callback functions are derived from a programming paradigm known as functional programming. They are very, very common in JavaScript. We'll use them a lot in the next section of the course, called "Handling events".

1. **Handling events**
   1. **Introduction**

Adding interactivity to a Web application can only be achieved with CSS, using the :hover pseudo CSS class, for instance. For example:





However, firing a specific action when the button is clicked, knowing which mouse button has been used, computing the (x, y) mouse pointer position in the button system coordinate, or executing more complex tasks can only be done through JavaScript.

With JavaScript, a button click, a move of the mouse, a resized window, and many other interactions create what are called "events".  The timing and order of events cannot be predicted in advance. We say that "event processing" is asynchronous. Web browsers detect events as they occur, and may pass them to JavaScript code. They do this by allowing you to register functions as event listeners, also called handlers or callbacks for specific events.

Each time an event occurs, the browser puts it in a "queue of events".

Then the browser looks at a list of "Event Listeners" and calls the ones that correspond to the type of event "they listen to".

* 1. **Adding and removing event listeners**

**[ -- Live coding video: adding an event listener to a document -- ]**

**[ -- Live coding video: adding an event listener to a specific HTML element-- ]**

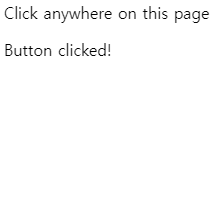
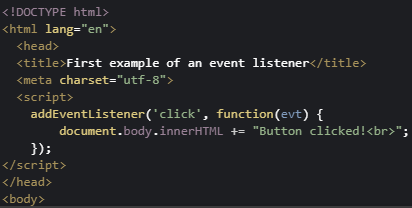
### **Adding and removing event listeners**

#### Event listeners: a typical example

Here is one possible syntax for registering an event listener that listens to "click" events on any part of the window (clicks anywhere on a web document will be processed by this event handler):

1. <script>
2. addEventListener('click', **function(evt) {**
3. **document.body.innerHTML += 'Button clicked!';**
4. **}**);
5. </script>

Try it below by clicking anywhere on the document:



The addEventListener function is one possible syntax for registering a function to be called when a given type of event occurs.

1. addEventListener(type\_of\_event, callback\_function)

In the example below, the type of event is a 'click', and the callback function is the part in bold:

1. function(evt) {
2. console.log("Button clicked!");
3. }t

When this function is small (a few lines of code), it's common practice to put its body as the second parameter of the addEventListener function.

In other words, this:

1. <script>
2. addEventListener('click', **function(evt) {**
3. **document.body.innerHTML += 'Button clicked!';**
4. **}**);
5. </script>

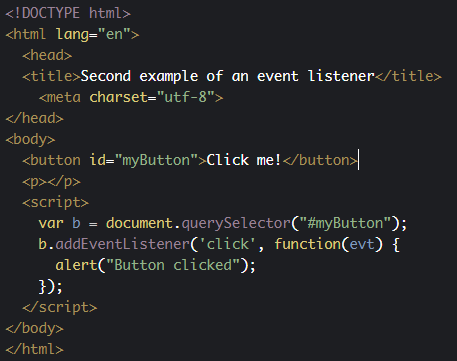
... is the same as this (the function called when a click occurs has its body "outside" of the addEventListener parameters, and we use its name as the second parameter):

1. <script>
2. addEventListener('click',**processClick**);
3. function**processClick**(evt) {
4. console.log("Button clicked!");
5. }
6. </script>

#### Adding an event listener to specific HTML elements

Instead of listening to event on the whole document (using addEventListener is the same as using window.addEventListener), we can listen to specific DOM elements.

For example, here is how we can listen to clicks on a specific button (whereas clicks on the rest of the document will be ignored).

****

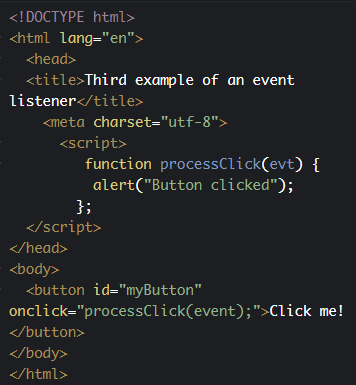
In this example, instead of using the addEventListener method directly, we used it on a DOM object (the button):

* 1. Get a reference of the HTML element that can fire the events you want to detect. This is done using the DOM API that we'll cover in detail later this week. In this example we used one of the most common/useful methods: var b = document.querySelector("#myButton");
  2. Call the addEventListener method on this object. In the example: b.addEventListener('click', callback)

Every DOM object has an addEventListener method. Once you get a reference of  any HMTL element from JavaScript, you can start listening to events on it.

#### An alternative method for adding an event listener to an HTML element: use an "on" attribute (ex: onclick = "....")

Instead of using b.addEventListener('click', callback), it's possible to use an onclick='doSomething();' attribute directly in the HTML tag of the element:

** **

This syntax:

1. <button id="myButton" **onclick="processClick(event);**">Click me!</button>

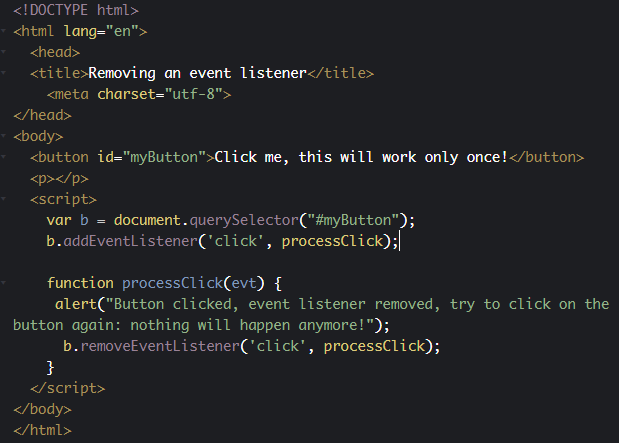
... is ok when you only need a single event listener to click events for this button, as there can be only one onclick attribute per element.

Using the b.addEventListener('click', callback) syntax,  you can register more than one event listener. You'll need rarely to do this, so in my opinion it's fine to choose whichever syntax you like.

Remember that for big projects, it's always better to separate the HTML, CSS and JavaScript code. In this case, I'd recommend that you put all your event listener definitions in a separate JavaScript file, and use the addEventListener syntax in preference to the "on" attributes syntax.

#### Removing event listeners

When we click on the button, we execute the processClick(evt) callback function, and inside we remove the listener we previously registered. Consequence: if we click on the button again, nothing happens as there is no longer a click event listener attached to it.

****

****

Note that to remove an event listener, you should have added it with its named function, so that we can pass it to both addEventListener and removeEventListener.

**Knowledge check 2.4.1 (not graded)**

**What precaution should you take when adding an event listener to a given HTML element?**

* I need to be sure that the element is in the DOM before selecting it using the DOM API or the selector API
* Nothing special
  1. **The event object**

### The event object

#### The event object is the only parameter passed to event listeners

Typical example:

1. function processClick(**evt**) {
2. alert("Button clicked!");
3. }

Each event listener has a single parameter that is  a "DOM event object". It has various properties and methods that can be very useful.

For example, with a 'keyup', 'keydown' or 'keypress' event, the event object contains the code of the key that has been pressed/released, with a 'mousemove' listener we can get the relative position of the mouse in the DOM element that has generated the event, etc.

The event object contains some important properties and methods that are common to all types of events:

* + **evt.type**: the name of the event
  + **evt.target:** for example, is the HTML element that has fired the event. In our previous examples with the click listeners on a button, **evt.target** in the event listener is the button itself.
  + **evt.stopPropagation()**: will not propagate the event to all other elements that listen to it. If several elements are registered for a click event - for example, you have a click listener on a button and on the window (the whole page). If you click on the button, and if in its click event listener you call evt.stopPropagation(); then the click event listener on the window object will never be called.
  + **evt.preventDefault()**: the default browser behavior will not be executed. For example, in a 'contextmenu' event listener attached to an object, if you call **evt.preventDefault()**, instead of having the right click default context menu of your browser displayed, you'll be able to display your own context menu, like [in this example](http://jsbin.com/kuyorac/edit?html,css,js,console,output).

It also contains properties that are associated with the type of the event, for example:

* + **evt.button**: the mouse button that has been used in the case of a mouse event listener
  + **evt.keyCode**: the code of the key that has been used
  + **evt.pageX**: coordinate of the mouse relative to the page
  + etc.

In the subsequent sections of this course we will look at the most common types of events in detail.

#### Reference table

The most useful common properties are:

|  |  |
| --- | --- |
| type | Returns the name of the event. |
| target | Returns the element that triggered the event. |

The most useful common methods are:

|  |  |
| --- | --- |
| preventDefault() | Cancels the event if it is "cancelable", meaning that the default action that belongs to the event will not occur. It is useful for cancelling the default browser behavior. For example: if you want to create a context menu that pops up with a right click, you must prevent the default behavior of the browser that will pop up its default context menu. |
| stopPropagation() | Prevents further propagation of an event during event flow. |

* 1. **Page lifecycle events**

**[ -- Live coding video: page 'load' event and the event object -- ]**

### **Page lifecycle events**

These events detect when the page is loaded and when the DOM is ready.

#### Events related to the page lifecycle

There are many other events related to the page life cycle. The most useful ones for an introduction course are shown below:

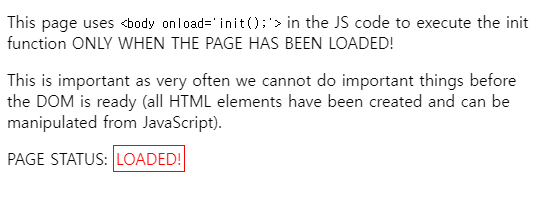
|  |  |
| --- | --- |
| load | This event occurs when an object has loaded (including all its resources: images, etc.). This event is very useful when you want to run JS code and be sure that the DOM is ready (in other words, be sure that a document.getElementById(...) or document. querySelector(...) will not raise an error because the document has not been loaded and elements you are looking for are not ready). |
| resize | The event occurs when the document view is resized. Usually, we get the new size of the window inside the event listener using var w = window.innerWidth; and var h = window.innerHeight; |
| scroll | The event occurs when an element's scrollbar is being scrolled. Usually in the scroll event listener we use things such as:   var max = document.body.scrollHeight - innerHeight;  var percent = (pageYOffset / max); ...to know the percentage of the scroll in the page. |

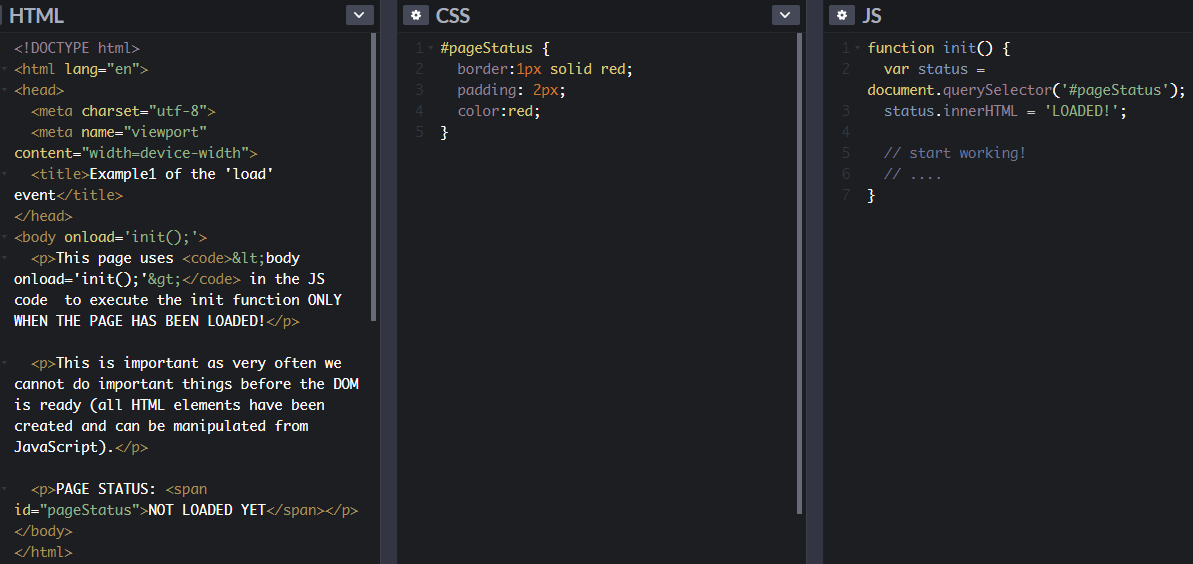
##### **Page event properties**

There are no particular properties that need to be mentioned here. Usually, the load event listener corresponds to a JavaScript function that can be seen as "the main" function of your Web application. It is a best practice to start everything after the page has been completely loaded. In the resize listener, you get the new size of the window, or the new size of some HTML elements in the page (as they might have been resized too when the window was resized), and then you do something (redraw a graphic in an HTML canvas that takes into account the new canvas size, for example).

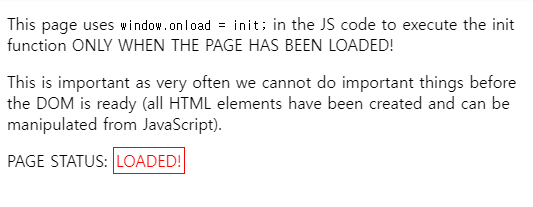
#### Example 1: wait until the page is loaded (when the DOM is ready) before doing something

This first variant that uses <body onload="init();">



****

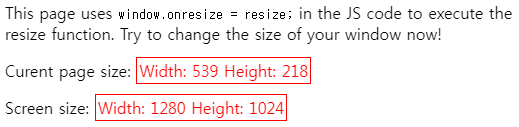
This second variant: using window.onload = init; in the JavaScript code...

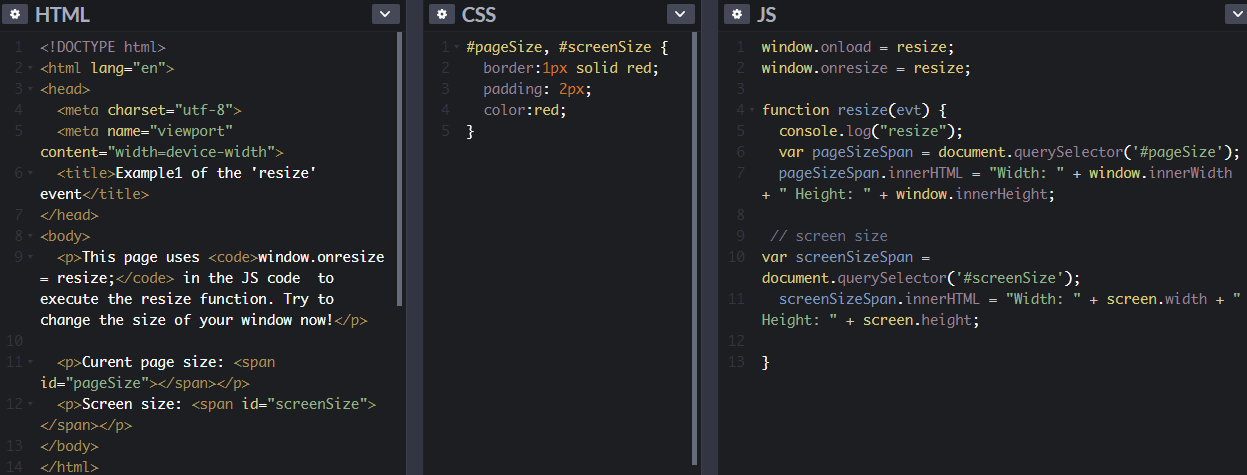
****

****

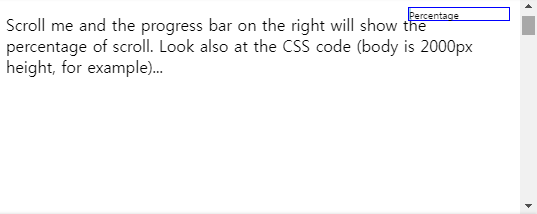
#### Example 2: detect a resize of the window

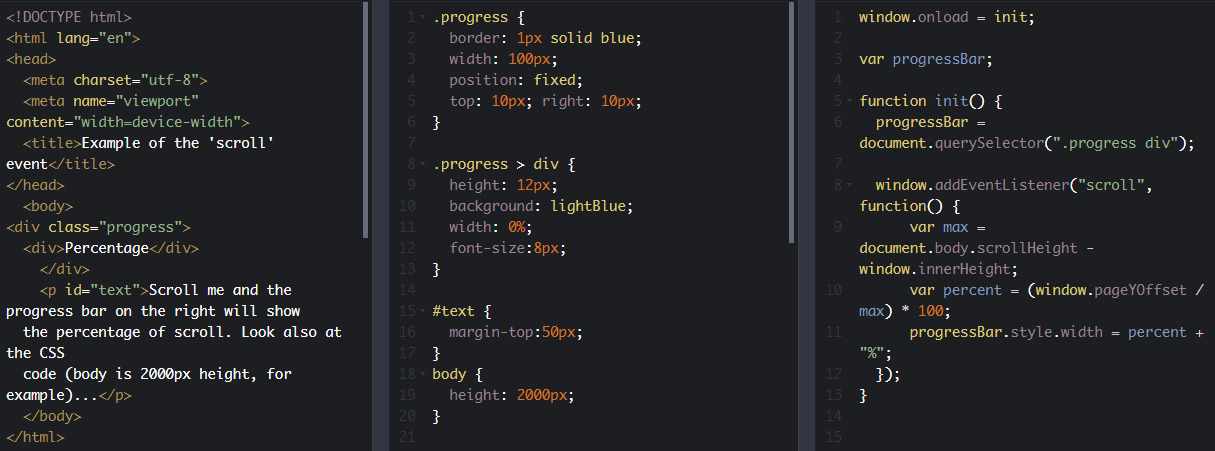
In this example, we're listening to page load and page resize events. When the window is loaded for the first time, or resized, we call the resize() callback function. The window.innerWidth and window.innerHeight properties are used to display the updated size of the window. We also use screen.width and screen.height to display the screen size.





#### Example 3: do something as the page is being scrolled up or down



****

**Knowledge check 2.4.2 (not graded)**

function init(evt) {

console.log("Page loaded! DOM Ready!");

// access the DOM using the DOM API or the selector API

var elem = document.querySelector(...);

elem.innerHTML = ....;

**Check the correct ways to call the function init only when the page has loaded and the DOM is ready:**

1. <body onload="init();"/>
2. In a JS code, add window.onload = init;
3. in a JS code, add window.addEventListener('load', init);
   1. **Key events: legacy API**

### **Dealing with key events**

#### picture of a keyboardIntroduction

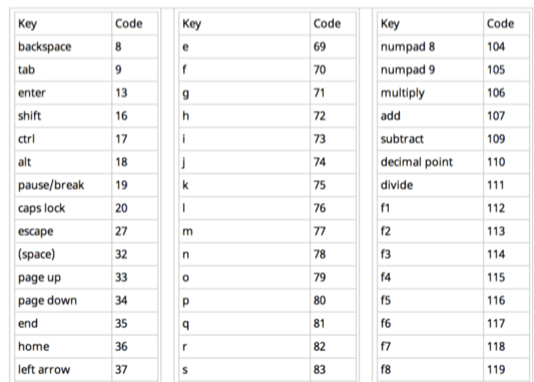
This has been a bit of a nightmare for years, as different browsers have had different ways of handling key events and key codes ([read this](http://unixpapa.com/js/key.html) if you are fond of JavaScript archeology). Fortunately it's much better today, and we are able to rely on methods that should work on any browser.

When you listen to keyboard related events (keydown, keyup or keypressed), the event parameter passed to the listener function will contain the code of the key that fired the event. Then it is possible to test which key has been pressed or released, like this:

1. window.addEventListener('keydown', function(event) {
2. if (**event.keyCode === 37**) {
3. //left arrow was pressed
4. }
5. });

At line 2, the value "37" is the key code that corresponds to the left arrow. It might be difficult to know which codes represent which real keyboard keys, so here are some handy pointers:

* Try key codes with this interactive example: <http://www.asquare.net/javascript/tests/KeyCode.html>
* And find a list of keyCodes (taken from: <http://css-tricks.com/snippets/javascript/javascript-keycodes/>) below:





#### The different key events

##### Event types related to keyboard

|  |  |
| --- | --- |
| keydown | The event occurs when the user is pressing a key. |
| keyup | The event occurs when the user releases a key. |
| keypress (now deprecated) | The event occurs when the user presses a key (up and release). |

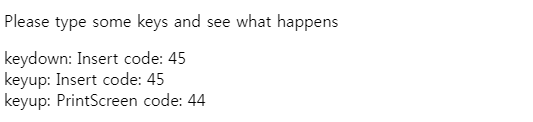
##### keyboardEvent properties

These are legacy properties, still used by many JavaScript code around the world. However, we do not recommend that you use them if you are targeting modern browsers. keyCode has a more powerful/easy to use replacement called code (not yet supported by all browsers), that comes with a new key property (see the following pages of the course).

|  |  |
| --- | --- |
| keyCode | Returns the Unicode character code of the key that triggered the onkeypress ,onkeydown or onkeyup event. |
| shiftKey | Returns whether the "shift" key was pressed when the key event was triggered. |
| ctrlKey | Returns whether the "ctrl" key was pressed when the key event was triggered. |
| altKey | Returns whether the "alt" key was pressed when the key event was triggered |

#### Example 1: use keyup and keydown on the window object

****

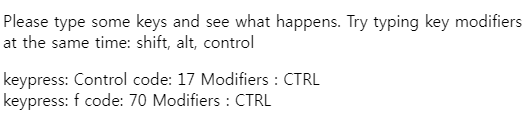
****

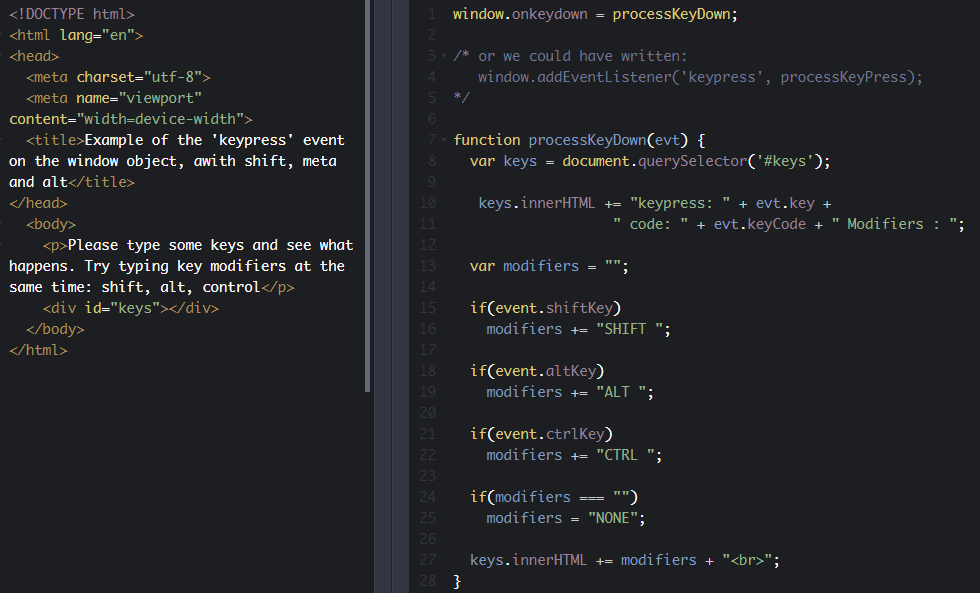
#### Example 2: see  keypress on the window object

See the Pen [keyup and keydown events on window](http://codepen.io/w3devcampus/pen/WpWjey/) by W3Cx ([@w3devcampus](http://codepen.io/w3devcampus)) on [CodePen](http://codepen.io/).

#### Example 3: detect a combination of keys + modifier keys (shift, ctrl, alt)

Try to type shift-a for example, ctrl-shift-b or alt-f...





**Knowledge check 2.4.3 (not graded)**

**Which of one of these events is now deprecated?**

1. keypress
2. keydown
3. keyup
   1. **Dealing with different keyboard layouts**

### Internationalize your keyboard controls

#### Introduction

Please do not assume that each key is at the same location on the keyboard in every country!

We've shown how to detect keyup, keydown and keypress events using the DOM API, and how to use the keyCode property of the DOM event.

Be careful when you use the key events in your application, as keyboard layouts vary from one country to another. Most first person shooter games (FPS) use three keys located on the top left of your keyboard to move your character. French AZERTY keyboards will use ZQSD for this (Z = up/move forward, Q and D are for left/right and S is for down/move backward), while US keyboards will use WASD, for example. So keep in mind that keys are not located at the same place on keyboards from different countries.

Extract from the "[Internationalise your keyboard controls](https://hacks.mozilla.org/2017/03/internationalize-your-keyboard-controls/)" article on MDN, by Julien Wajsberg:

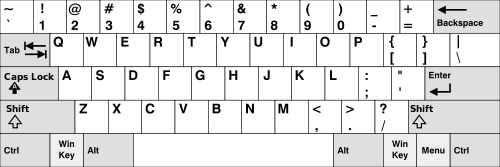
"Recently I came across two lovely new graphical demos, and in both cases, the controls would not work on my French [*AZERTY keyboard*](https://en.wikipedia.org/wiki/AZERTY).

There was the wonderful WebGL 2 technological demo [*After The Flood*](https://www.youtube.com/watch?v=TT7ugKuUMv0/), and the very cute [*Alpaca Peck*](http://codepen.io/shshaw/full/apwMwM/). [*Shaw*](http://codepen.io/shshaw/) was nice enough to fix the latter when I told him about the issue. It turns out the Web browser actually exposes a useful API for this."

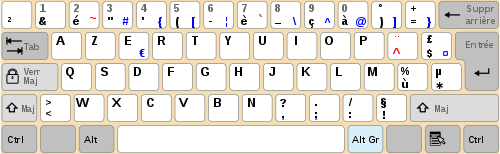
#### One keyboard, many layouts

For details, see [Wikipedia’s keyboard layout page](https://en.wikipedia.org/wiki/Keyboard_layout)!

##### [QWERTY](https://en.wikipedia.org/wiki/QWERTY) layout, used in US, GB, etc.:

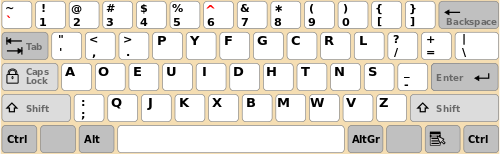


##### AZERTY layout, used in some French-speaking countries:



##### In addition, QWERTZ keyboards are in use in Germany and other European countries, and DVORAK is another alternative to QWERTY:

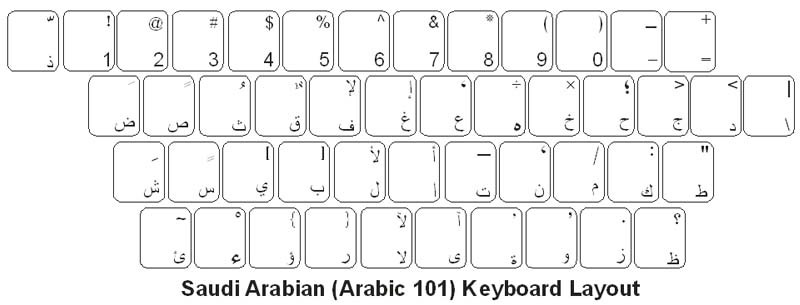
DVORAK:



QWERTZ:



##### Saudi Arabic keyboard layout:



##### Bangla National (Jatiyo) keyboard:



* 1. **Key and code properties**

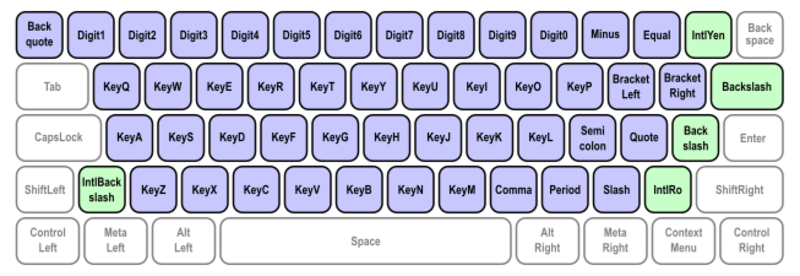
### New recommended properties you can use with modern browsers: key and code

You may have noticed that in some examples from the previous course page about key events, we used event.key in order to display the character that has been typed. The key property has been introduced with a new W3C API called UI Events (or DOM level 3 events), that has been discussed since 2000.  All major browsers have implemented this very practical key property. It comes with another property named code, which is what keyCode should have been. The value of the code property corresponds to a code that is more readable than the value of the old keyCode property.

* **key:**when the pressed key is a printable character, you get the character in string form. When the pressed key is not a printable character (for example: Backspace, Control, but also Enter or Tab which actually are printable characters), you get a multi-character descriptive string, like 'Backspace', 'Control', 'Enter', 'Tab'.
* **code**: Gives you the physical key that was pressed, in string form. This means it’s totally independent of the keyboard layout that is being used. So let’s say the user presses the Q key on a QWERTY keyboard. **Then event.code gives you 'KeyQ' while event.key gives you 'q'.  
    
  But when an AZERTY keyboard user presses the A key, he also gets 'KeyQ' as event.code, yet event.key contains 'a'. This happens because the A key on a AZERTY keyboard is at the same location as the Q key on a QWERTY keyboard.**As for numbers, the top digit bar yields values like 'Digit1', while the numeric pad yields values like 'Numpad1'.  
    
  Unfortunately this feature is not yet implemented by Microsoft IE/Edge but support is coming soon to Edge.

#### List of codes, the reference keyboard

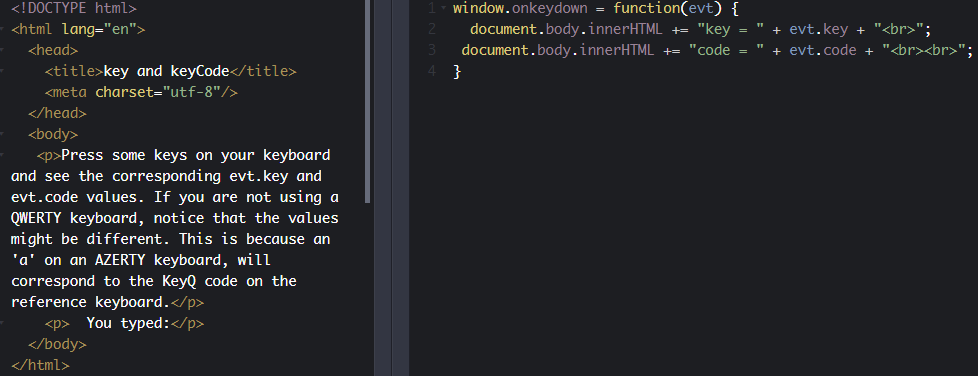
There’s no existing keyboard with all the possible keys. That’s why the W3C published [a specification just for this](https://www.w3.org/TR/uievents-code/). You can read about the [existing mechanical layouts](https://www.w3.org/TR/uievents-code/#keyboard-layout) around the world, as well as their [reference keyboard](https://www.w3.org/TR/uievents-code/#code-value-tables). For instance here is their reference keyboard for the alphanumerical part:

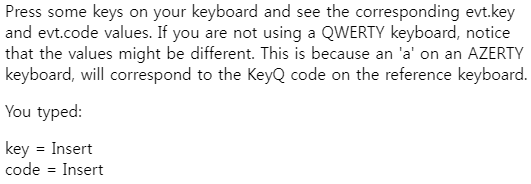


You can also read [this document published by the W3C with explanations about all the possible values for the code property](https://www.w3.org/TR/uievents-key/).

Also read through [the examples given in the specification](https://w3c.github.io/uievents/#code-examples). They show very clearly what happens when the user presses various types of keys, both for code and key.

#### Example that displays the key and code values with your current keyboard

****

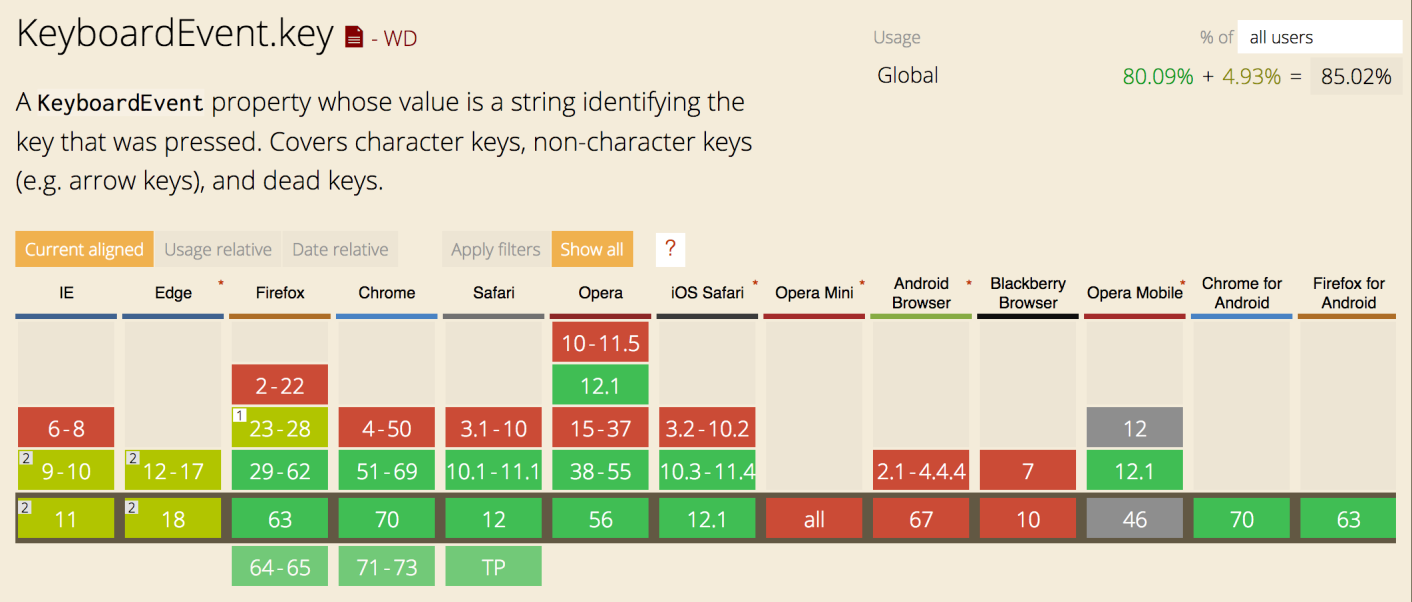
****

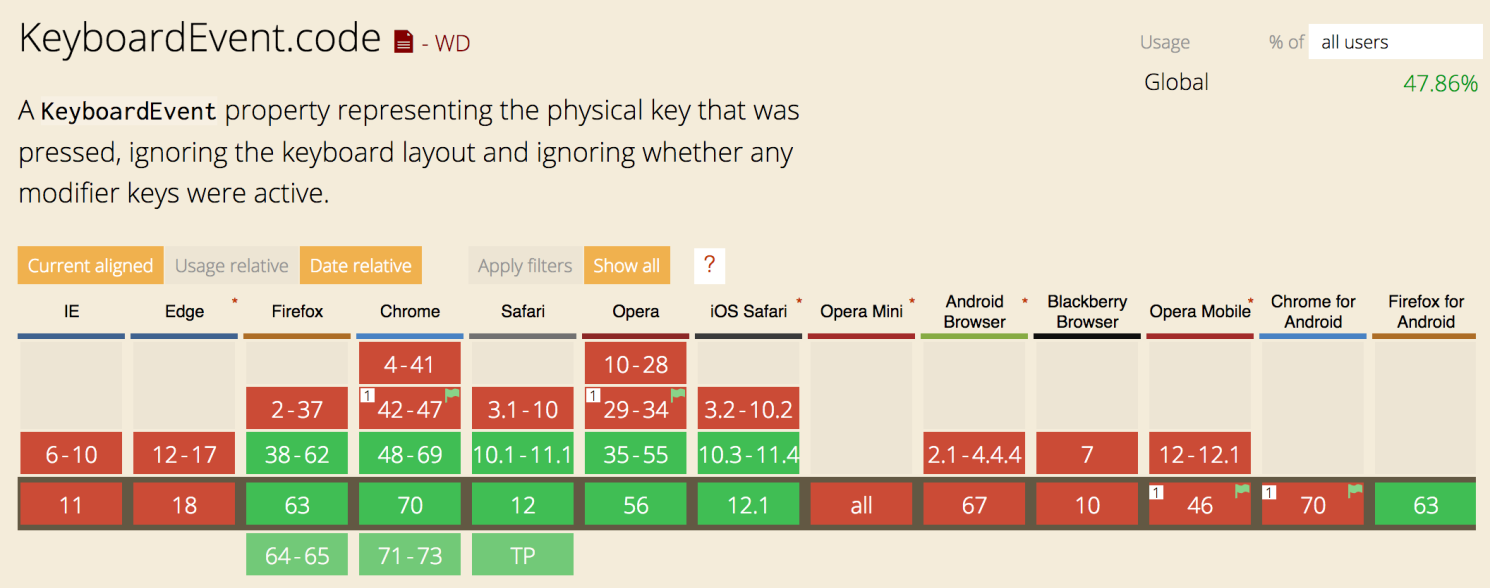
I encourage you to take a look and get at least an overview of this specification.

Please note that the W3C has also published a sibling specification describing the values for the key property.

#### Before looking at examples, let's see the current Web browser support for these properties:

As of December 2018:





* 1. **Mouse events**

### **Mouse interaction, mouse events**

#### Illustration of mouse events' typesIntroduction

***Important note***: Remember that many people do not use the mouse and rely on the keyboard to interact with the Web. This requires keyboard access to all functionality, including form controls, input, and other user interface components ([*learn more*](https://www.w3.org/WAI/intro/people-use-web/principles#operable)).

Detecting mouse events in a canvas is quite straightforward: you add an event listener to the canvas, and the browser invokes that listener when the event occurs.

The example below is about listening to mouseup and mousedown events (when a user presses or releases any mouse button):

1. canvas.addEventListener('mousedown', function (evt) {
2. // do something with the mousedown event
3. });
5. canvas.addEventListener('mouseup', function (evt) {
6. // do something with the mouseup event
7. });

The event received by the listener function will be used for getting the button number or the coordinates of the mouse cursor. Before looking at different examples, let's look at the different event types we can listen to.

#### Mouse events

##### Event types related to mouse

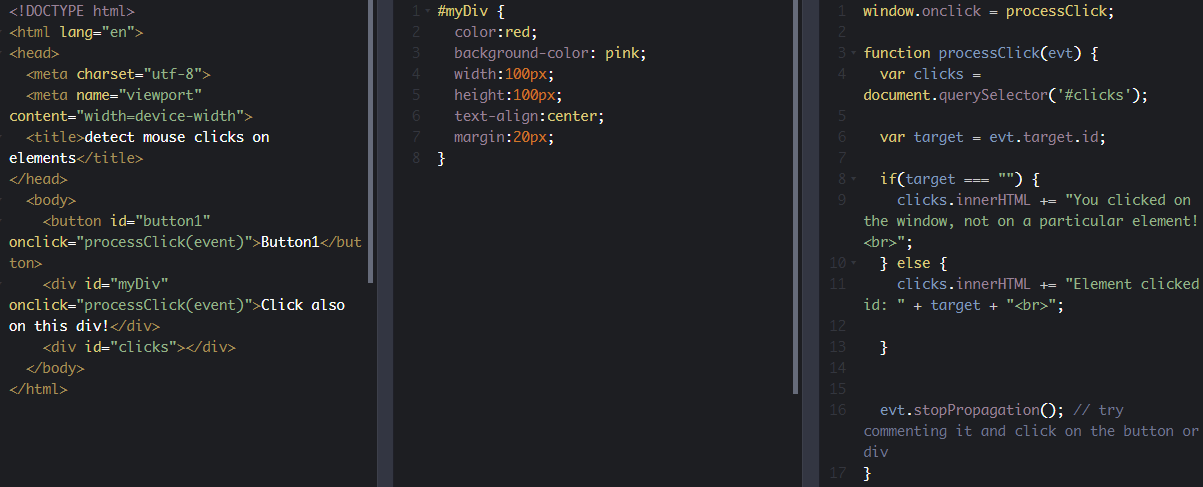
|  |  |
| --- | --- |
| click | The event occurs when the user clicks on an element (presses a button and releases it) |
| dblclick | The event occurs when the user double-clicks on an element |
| mousedown | The event occurs when the user presses a mouse button |
| mouseup | The event occurs when a user releases a mouse button over an element |
| mousemove | The event occurs when the pointer is moving while it is over an element |
| mouseenter | The event occurs when the pointer is moved onto an element |
| mouseleave | The event occurs when the pointer is moved out of an element |
| mouseover | The event occurs when the pointer is moved onto an element, or onto one of its children |
| contextmenu | The event occurs when the user right-clicks on an element to open a context menu |

##### MouseEvent properties

|  |  |
| --- | --- |
| button | Returns which mouse button was pressed when the mouse event was triggered |
| clientX and clientY | Returns the coordinates of the mouse pointer, relative to the element coordinate system that triggered the event. **If you click in the left top corner the value will always be (0,0) independent of scroll position, these coordinates are relative to the VIEWPORT (the visible part of the document page)** |
| pageX and pageY | Returns the coordinates of the mouse pointer, relative to the document, when the mouse event was triggered. **They are relative to the complete document/page, and will always be relative to the very beginning of the document/page, even if the top of the page is not visible because you've scrolled down. They will change when the page scrolls and the mouse does not move!** |
| screenX and screenY | Returns the coordinates of the mouse pointer, relative to the screen, when an event was triggered. |
| altKey, ctrlKey, shiftKey | Returns whether the "alt, ctrl and shif" key was pressed when an event was triggered |
| detail | Returns a number that indicates how many times the mouse was clicked |

#### Examples

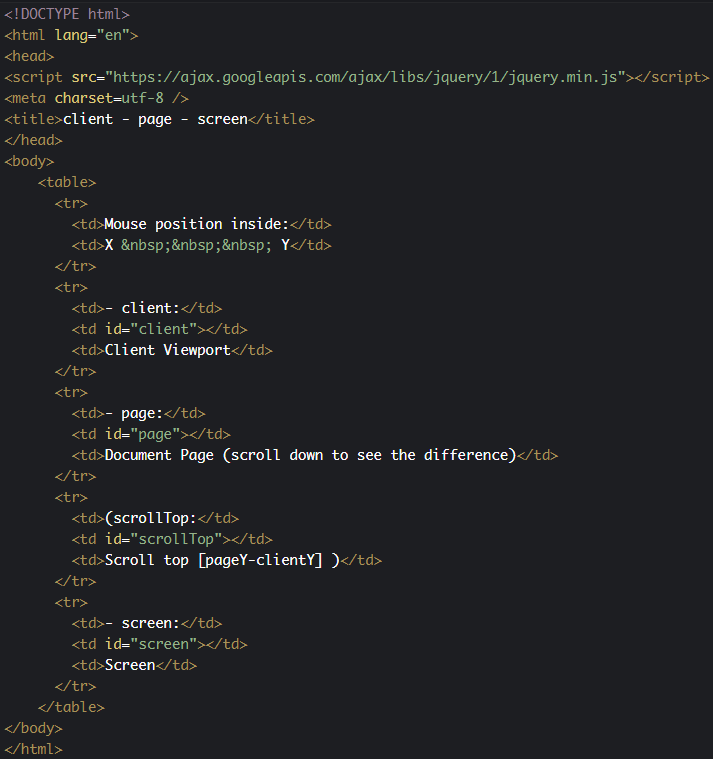
##### Example 1: detect a click on an element

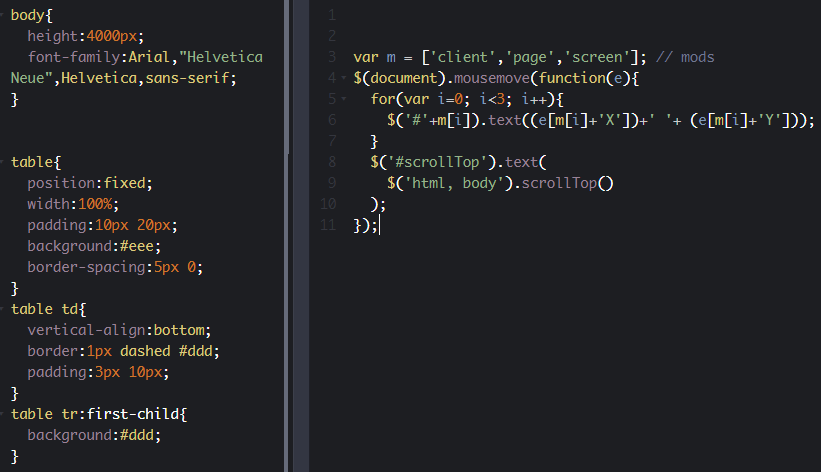
****

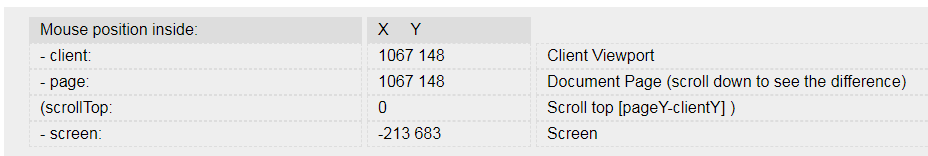
****

##### **Example 2: see the differences between clientX/clientY and pageX/pageY**

The source code is not meant to be understood. It uses the jQuery lib and comes from [a stackOverflow thread](http://stackoverflow.com/questions/9262741/what-is-the-difference-between-pagex-y-clientx-y-screenx-y-in-javascript). But it's interesting: move the mouse pointer, look at the different properties. Then scroll the page (the table at the top will not move), and look at the properties again; notice how pageX/pageY change, since they are relative to the top of the page, even if they are not visible. Click on the codePen label on top right and run this example! It does not work when embedded in this page!



****

****

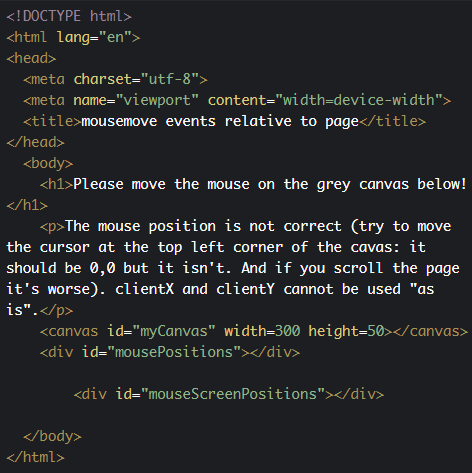
##### **Example 3: detect a mousemove event and get the mouse position relative to the page**

****

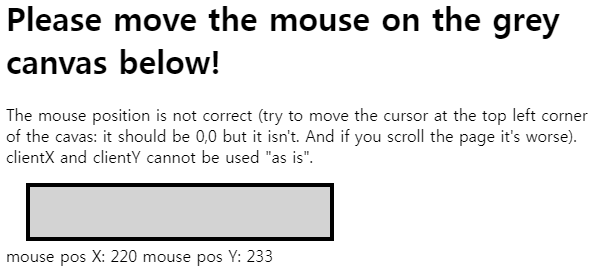
****

##### **Example 4: detect a mousemove and get the mouse position relative to the element that fired the event**

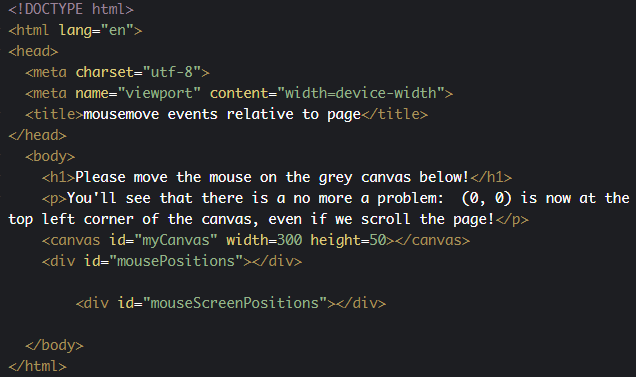
Here is a first version that does not work well due to a naive use of clientX/PageX and clientY/pageY mouse event properties:

****

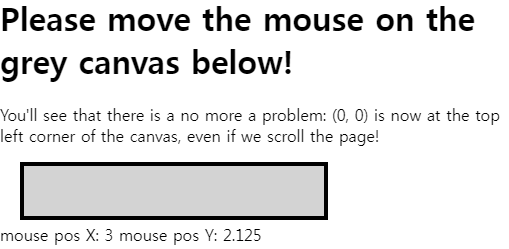
****

****

Here is another version that uses client / clientY and the e.target.getBoundingClientRect () method that returns the bounding rectangle that contains the element that fired the event. The return value has top, left, width, and height properties that describe this rectangle. We can use the top and left properties along with evt.clientX and evt.clientY to fix the mouse position and to get a real position relative to the top left corner of the canvas:

****

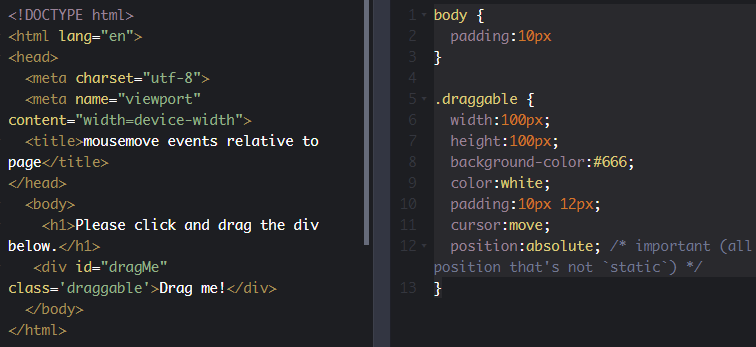
****

****

JavaScript source code extract:

1. function processMouseMouve(evt) {
2. var mousePositions = document.querySelector('#mousePositions');
3. // adjust mouse position relative to the canvas
4. **var rect = evt.target.getBoundingClientRect()**
5. **var mouseX = evt.clientX - rect.left;**
6. **var mouseY = evt.clientY - rect.top;**
7. mousePositions.innerHTML = "mouse pos X: " + mouseX +
8. " mouse pos Y: " + mouseY +
9. "<br>"
10. }

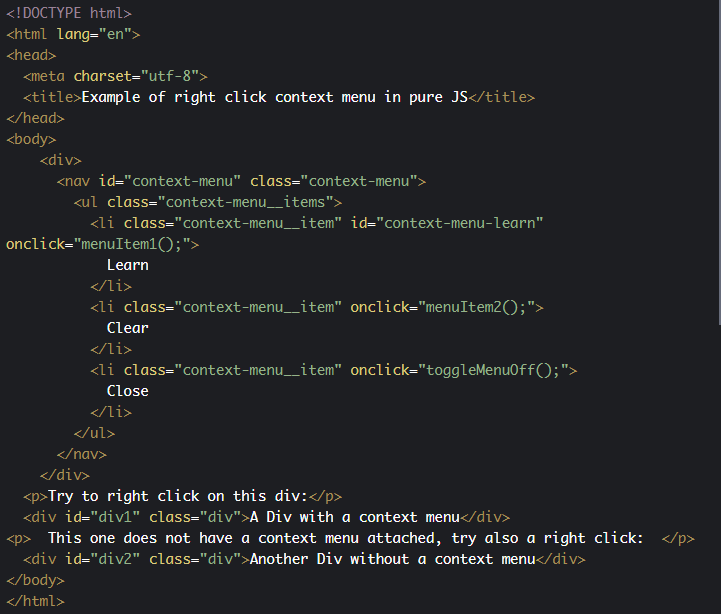
##### Example 5: combine mouseup, mousedown, mousemove to implement a click and drag behavior

****

|  |
| --- |
| <script>    window.onmousemove = moveElem;    window.onmouseup = stopMovingElem;    window.onload = init;    var selected = null; // element to be moved    var oldMouseX, oldMouseY; // Stores x & y coordinates of the mouse pointer    var elemX, elemY;    function init() {      document.querySelector('.draggable').onmousedown = function (evt) {        dragInit(evt);      };    }    // Will be called when user starts dragging an element    function dragInit(evt) {      // Store the elem      selected = evt.target;      elemX = selected.offsetLeft;      elemY = selected.offsetTop;      oldMouseX = evt.clientX;      oldMouseY = evt.clientY;    }    // Will be called when user dragging an element    function moveElem(e) {      // new mouse ps      var newMouseX = e.clientX;      var newMouseY = e.clientY;      if (oldMouseX !== undefined) {        // how many pixels did we move the mouse?        var dx = newMouseX - oldMouseX;        var dy = newMouseY - oldMouseY;      }      if (selected !== null) {        // move the selected element dx, dy pixels hozontally/vertically        changePosOfSelectedElement(dx, dy);      }      // update the old position of the mouse      oldMouseX = newMouseX;      oldMouseY = newMouseY;    }    function changePosOfSelectedElement(dx, dy) {      // update the old position of the selected element      elemX += dx;      elemY += dy;      // change the pos on screen of the element      // by modifying its CSS left/top properties      selected.style.left = elemX + 'px';      selected.style.top = elemY + 'px';    }    // Destroy the object when we are done    function stopMovingElem() {      selected = null;    }  </script> |

****

##### **Example 6: create  and attach a right-click context menu to any element**

****

|  |
| --- |
| <style>    /\* CONTEXT MENU \*/    /\* context menu \*/    .context-menu {      display: none;      position: absolute;      z-index: 10;      padding: 12px 0;      width: 240px;      background-color: #fff;      border: solid 1px #dfdfdf;      box-shadow: 1px 1px 2px #cfcfcf;    }    .context-menu--active {      display: block;    }    .context-menu\_\_items {      list-style: none;      margin: 0;      padding: 0;    }    .context-menu\_\_item {      display: block;      margin-bottom: 4px;      padding: 4px 12px;      color: #0066aa;      text-decoration: none;    }    .context-menu\_\_item:last-child {      margin-bottom: 0;    }    .context-menu\_\_item:hover {      color: #fff;      background-color: #0066aa;    }    #div1 {      background-color: red;      height: 100px;    }    #div2 {      background-color: green;      height: 100px;    }  </style> |
| <script>    window.onload = init;    var menu, menuIsVisible;    function init() {      menu = document.querySelector("#context-menu");      menuIsVisible = false;      /\* to attach a context menu to all divs, you can do this:         var divs = document.querySelectorAll(".div");           divs.forEach(function(d) {              addContextMenu(d);         });      \*/      // attache the context menu to the first div      var div1 = document.querySelector("#div1");      addContextMenu(div1);      // Clicking anywhere on the window toggle the menu off      window.addEventListener('click', toggleMenuOff);    }    function addContextMenu(elem) {      elem.addEventListener("contextmenu", function (e) {        //console.log("contextmenu activated");        e.preventDefault(); // avoids default right click menu        toggleMenuOn();        positionMenu(e);      });    }    function toggleMenuOn() {      if (!menuIsVisible) {        menuIsVisible = true;        menu.classList.add("context-menu--active"); // see further in the DOM section of the course      }    }    function toggleMenuOff() {      if (menuIsVisible) {        menuIsVisible = false;        menu.classList.remove("context-menu--active");      }    }    function positionMenu(e) {      // Mouse position is relative to the element clicked      // We make the coords absolute in the page      var clickCoordsX = e.pageX;      var clickCoordsY = e.pageY;      var menuWidth = menu.offsetWidth + 1;      var menuHeight = menu.innerHeight + 1;      var elementWidth = e.target.offsetWidth;      var elementHeight = e.target.offsetHeight;      if ((elementWidth - clickCoordsX) < menuWidth) {        menu.style.left = elementWidth - menuWidth + "px";      } else {        menu.style.left = clickCoordsX + "px";      }      if ((elementHeight - clickCoordsY) < menuHeight) {        menu.style.top = elementHeight - menuHeight + "px";      } else {        menu.style.top = clickCoordsY + "px";      }    }    // Actions called when a menu item is choosen    function menuItem1() {      console.log('learn');      toggleMenuOff();    }    function menuItem2() {      console.log('clear');      toggleMenuOff();    }  </script> | |

****

* 1. **Form and input field events**

#### Forms

##### Events related to forms

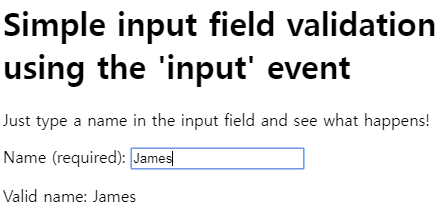
|  |  |
| --- | --- |
| input | The event occurs when an element gets user input (e.g., a key is typed on an input field, a slider is moved, etc.) |
| change | The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <select>, and <textarea>). A change event listener on a slider will generate an event when the drag/move ends, while input events will be useful to do something as the slider is being moved. |
| focus | The event occurs when an element gets focus (e.g., the user clicks in an input field) |
| blur | The event occurs when an element loses focus (e.g., the user clicks on another element) |
| select | The event occurs after the user selects some text (for <input> and <textarea>) |
| submit | The event occurs when a form is submitted |

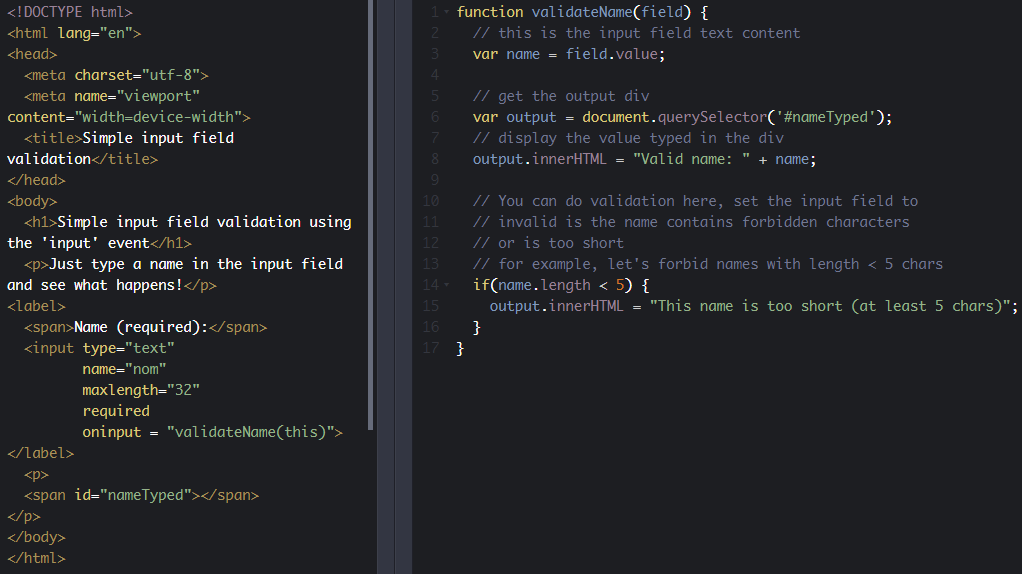
##### FormEvent properties

There are no particular properties that need to be mentioned here. Usually, on a form event listener, we check the content of the different input fields, using their value property. See examples in the part of the course that deals with form events.

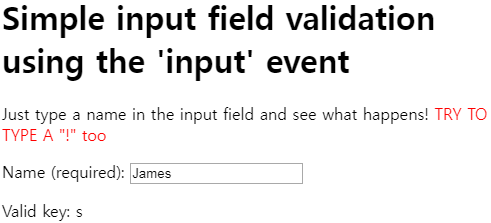
#### Example 1: validating on the fly as the user types in a text input field

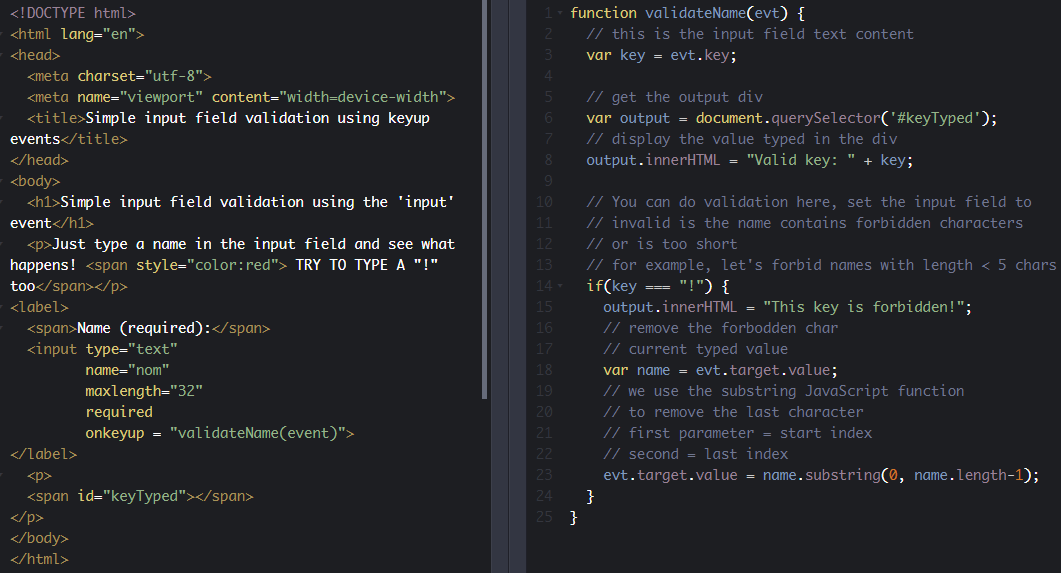
First variant: using the 'input' event:



****

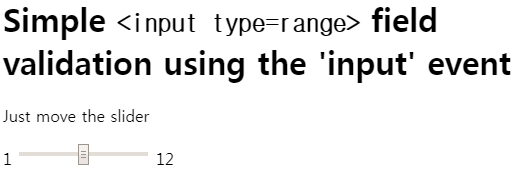
Second variant: using the 'keyup' event:

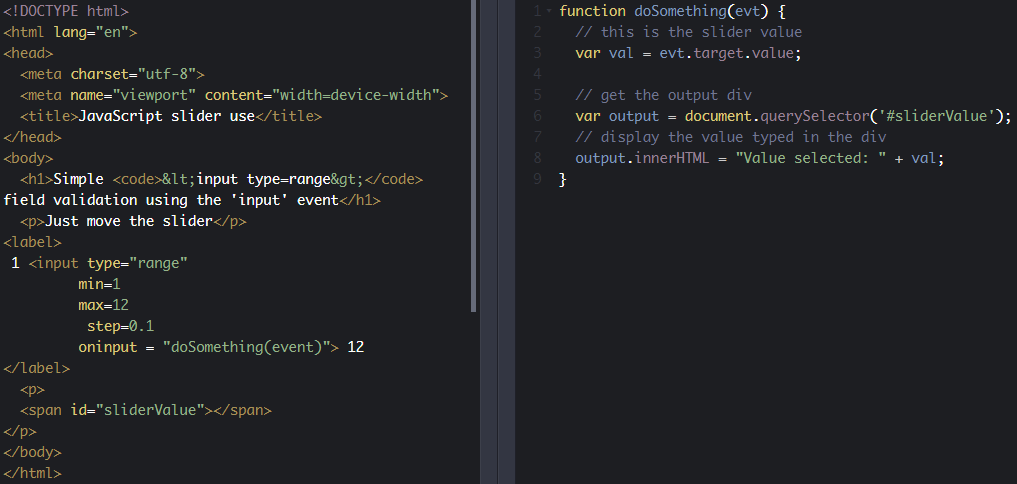


****

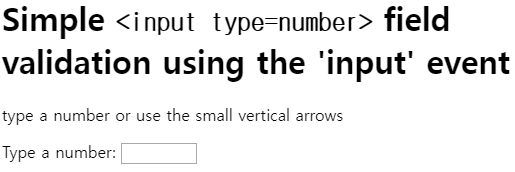
Note that HTML5 forms and the multiple facets of form and input field validation are covered in depth in the W3C [HTML5 Coding Essentials and Best Practices](https://www.edx.org/course/html5-coding-essentials-and-best-practices) course, which dedicates a whole week to this topic.

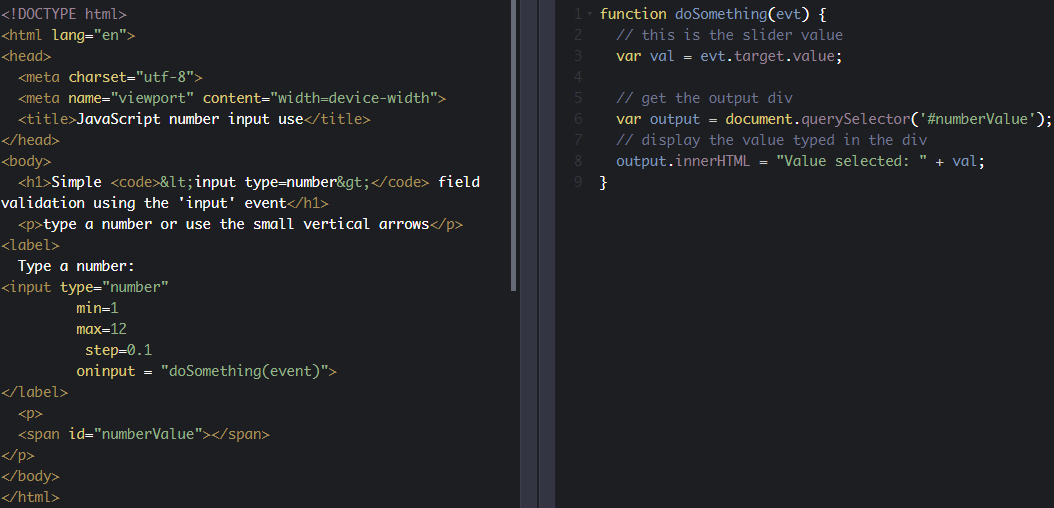
#### Example 2: do something while a slider is being moved



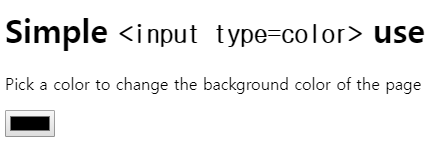


#### Example 3: detect value changes in a number input field



****

#### Example 4: choose a color and do something

****

****

* 1. **Reference tables**

### **Quick summary of event management in JavaScript**

#### HTML5 EVENTS

There is no input or output in JavaScript. We treat events caused by user actions as inputs, and we manipulate the DOM structure as output. Usually in a JavaScript application, we will get info such as the key strokes, the mouse button clicks and the mouse position, and we will refer to these variables when determining what action to perform.

In any case, the events are called DOM events, and we use the DOM APIs to create event handlers.

#### HOW TO LISTEN TO EVENTS

There are three ways to manage events in the DOM structure. You could attach an event inline in your HTML code like this:

##### **Method 1: declare an event handler in the HTML code**

1. <div id="someDiv" **onclick**="alert('clicked!')"> content of the div </div>

This method is very easy to use, but it is not the recommended way to handle events. Indeed, although it currently works, it is deprecated (will probably be abandoned in the future). Mixing 'visual layer' (HTML) and 'logic layer' (JavaScript) in one place is really bad practice and causes a host of problems during development.

##### **Method 2: attach an event handler to an HTML element in JavaScript**

1. document.getElementById('someDiv').**onclick**= function() {
2. alert('clicked!');
3. }

This method is fine, but  you will not be able to attach multiple listener functions. If you need to do this, use the version shown below.

##### **Method 3: register a callback to the event listener with the addEventListener method (preferred  method)**

1. document.getElementById('someDiv').**addEventListener**('click', function() {
2. alert('clicked!');
3. }, false);

Note that the third parameter describes whether the callback has to be called during the captured phase. This is not important for now, just set it to false or ignore it (you can even pass only two parameters to the addEventListener function call and do not set this boolean parameter at all).

#### Details of the DOM event are passed to the event listener function

When you create an event listener and attach it to an element, the listener will create an event object to describe what happened. This object is provided as a parameter of the callback function:

1. element.addEventListener('click', function(**event**) {
2. **// now you can use event object inside the callback**
3. }, false);

Depending on the type of event you are listening to, you will consult different properties from the event object in order to obtain useful information such as: "which keys are pressed down?", "what is the location of the mouse cursor?", "which mouse button has been clicked?", etc.

In the following lessons, we will remind you how to deal with the keyboard and the mouse.

### **Further reading**

In Method 1 (above), we mentioned that "mixing 'visual layer' (HTML) and 'logic layer' (JavaScript) ... is bad practice", and this is similarly reflected in many style features being deprecated in HTML5 and moved into CSS3. The management philosophy at play here is called "the separation of concerns" and applies in several ways to software development - at the code level, through to the management of staff. It's not part of the course, but professionals may find the following references useful:

* [Separation of concerns - Wikipedia, the free encyclopedia](https://en.wikipedia.org/wiki/Separation_of_concerns)
* [The Art of Separation of Concerns](http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/) by derekgreer, January 3, 2008

### **Reference tables for events and properties/methods**

These tables are provided as a reference. They are a compilation of the most common event types sorted by domain (key, mouse, forms, etc.). For each domain you will see the most useful event types and their properties.

In the following sections, we will show  examples that use most of the events displayed in these tables.

#### Event object

Most useful common properties:

|  |  |
| --- | --- |
| type | Returns the name of the event |
| target | Returns the element that triggered the event |

Most useful common methods:

|  |  |
| --- | --- |
| preventDefault() | Cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur |
| stopPropagation() | Prevents further propagation of an event during event flow |

#### Page

##### Events related to the page lifecycle

There are many other events related to the page life cycle. Below are the most useful ones for an intro course:

|  |  |
| --- | --- |
| load | This event occurs when an object has loaded (including all its resources: images, etc.) This event is very useful when you want to run JS code and be sure that the DOM is ready (in other words, be sure that a document.getElementById(...) or document.querySelector(...) will not raise an error because the document has not been loaded and elements you are looking for are not ready. |
| resize | The event occurs when the document view is resized. Usually we get the new size of the window inside the event listener using var w = window.innerWidth; and var h = window.innerHeight; |
| scroll | The event occurs when an element's scrollbar is being scrolled. Usually, in the scroll event listener, we use things such as:   var max = document.body.scrollHeight - innerHeight;  var percent = (pageYOffset / max); ...to know the percentage of the scroll in the page. |

##### Page event properties

There are no particular properties that need to be mentioned here. Usually, the load event listener corresponds to a JavaScript function that can be seen as "the main" function of your Web Application. It is good practice to start everything after the page has been completely loaded. In the resize listener, you get the new size of the window, or the new size of some HTML elements in the page (as they might have been resized too when the window was resized) and then you do something (redraw a graphic in an HTML canvas that takes into account the new canvas size, for example).

#### Keyboard

##### Event types related to keyboard

|  |  |
| --- | --- |
| keydown | The event occurs when the user is pressing a key |
| keyup | The event occurs when the user releases a key |
| keypress | The event occurs when the user presses a key (up and release) |

##### keyboardEvent properties

|  |  |
| --- | --- |
| keyCode | Returns the Unicode character code of the key that triggered the onkeypress ,onkeydown or onkeyup event |
| shiftKey | Returns whether the "shift" key was pressed when the key event was triggered |
| ctrlKey | Returns whether the "ctrl" key was pressed when the key event was triggered |
| altKey | Returns whether the "alt" key was pressed when the key event was triggered |

#### Mouse

##### Event types related to mouse

|  |  |
| --- | --- |
| click | The event occurs when the user clicks on an element (presses a button and releases it) |
| dblclick | The event occurs when the user double-clicks on an element |
| mousedown | The event occurs when the user presses a key (up and release) |
| mouseup | The event occurs when a user releases a mouse button over an element |
| mousemove | The event occurs when the pointer is moving while it is over an element |
| mouseenter | The event occurs when the pointer is moved onto an element |
| mouseleave | The event occurs when the pointer is moved out of an element |
| mouseover | The event occurs when the pointer is moved onto an element, or onto one of its children |
| contextmenu | The event occurs when the user right-clicks on an element to open a context menu |

##### MouseEvent properties

|  |  |
| --- | --- |
| button | Returns which mouse button was pressed when the mouse event was triggered |
| clientX and clientY | Returns the coordinates of the mouse pointer, relative to the element coordinate system that triggered the event |
| pageX and pageY | Returns the coordinates of the mouse pointer, relative to the document, when the mouse event was triggered |
| screenX and screenY | Returns the coordinates of the mouse pointer, relative to the screen, when an event was triggered |
| altKey, ctrlKey, shiftKey | Returns whether the "alt, ctrl and shift" key was pressed when an event was triggered |
| detail | Returns a number that indicates how many times the mouse was clicked |

#### Forms

##### Events related to forms

|  |  |
| --- | --- |
| input | The event occurs when an element gets user input (e.g., a key is typed on an input field, a slider is moved, etc.) |
| change | The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <select>, and <textarea>). A change event listener on a slider will generate an event when the drag/move ended, while input events will be useful to do something as the slider is being moved. |
| focus | The event occurs when an element gets focus (e.g., the user clicks in an input field) |
| blur | The event occurs when an element loses focus (e.g., the user clicks on another element) |
| select | The event occurs after the user selects some text (for <input> and <textarea>) |
| submit | The event occurs when a form is submitted |

##### FormEvent properties

There are no particular properties that need to be mentioned here. Usually, on a form event listener, we check the content of the different input fields, using their value property. See examples in the course, in the part dealing with form events.

* 1. **Discussion and project**

Here is the discussion forum for this part of the course. Please either post your comments/observations/questions or share your creations.

See below for suggested topics of discussion and an optional project to help you practice.

#### Suggested topics

* Did you know the differences between clientX, pageX, etc. when dealing with mousemove events? Did you know the method we showed for getting the mouse position relative to the elements you're moving the mouse on?
* Did you know the different keyboard layouts, and the code and key event properties?

#### Optional project

* Try to write a small game that will display a word taken from an array, randomly, and you will have to type the letters of this word as fast as you can. A bit like the "type of the dead" game (look on YouTube). You can imagine the evolution such as choosing the level, start with 3 letter words, 4 letter words, etc.  
  Choose a topic (medicine, with very difficult words to spell, etc.). If you look at the end of the module, you will see how to work with graphics and animation, you can also imagine a graphic version of this

1. **The DOM API**
   1. **Introduction**

### **The DOM standard - exploring the DOM of a document**

#### Introduction

When a user clicks on a link or enters a URL in the address of your Web browser, it downloads the page’s HTML text and builds up a model of the document’s structure called the DOM (Document Object Model). This model is used to render the HTML page on the screen.

The DOM is a standard  that describes how a document must be manipulated. It defines a “language- and platform neutral interface”. So, **every browser offers the same JavaScript DOM API**.

The DOM API is a programming interface the JavaScript programmer can use to modify the HTML content or the CSS style of HTML elements on the fly.

The DOM API provides the document object as a structured object, a group of nodes represented as a tree. We saw this in Module 1 when we revised the basic principles of HTML .

The document object also exposes a large set of methods to access and manipulate the structured document. Through the DOM, look for nodes (html elements that compose the page), move nodes, delete nodes, modify nodes (attributes, content), and also handle their associated events.

In JavaScript, the DOM is accessible through the property document of the global object window. We rarely manipulate the window object directly as it is implicit: window.document is the same as document.

So by using this object, we can access and manipulate our page from JavaScript as a structured document.

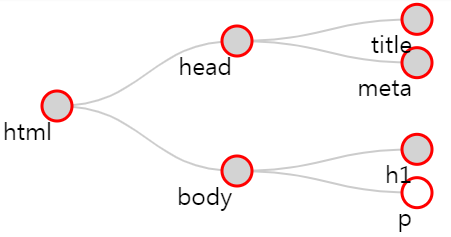
#### Reminder from Module 1: HTML and the DOM

'Elements' are the pieces themselves, i.e., a paragraph, a header, and even the body are elements. Most elements can contain other elements - for example, the body element would contain header elements, paragraph elements, in fact pretty much all of the visible elements of the Document Object Model (developers call it the "DOM").

Let's take, for example, a simplified version of the last HTML code we showed you:

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <title>Your first HTML page</title>
5. <meta charset="utf-8">
6. </head>
7. <body>
8. <h1>My home page</h1>
9. <p>Hi! Welcome to my Home Page! My name is Michel Buffa,
10. I'm a professor at the University of Nice, in France,
11. and I'm also the author of two MOOCS about HTML5 on
12. W3Cx.
13. </p>
14. </body>
15. </html>

Click the red circle next to HTML to unfold this HTML document structure (we can also say "see its DOM structure"):

****

Consider the figure above.  It contains a single html element.  It turns out this includes within it the entire content of your html file.  If you click on the "html" red node, you'll find that it contains two components, a head and a body.  Clicking on each of these will reveal their respective contents.  This structure is what we computer scientists call a "tree".  Any given element (except for the outermost 'html' element) is wholly contained inside another element, referred to as the "parent" element.  Not surprisingly, the elements that a given element contains are its "child" elements.  And, yes, children of a common parent are often referred to as "siblings".

Thus in the example above, the top element is the html element, which contains just two elements, the head and body.  The head element contains a title element and the body contains an h1 element and a p element.  In a more typical example, the body would contain many more children, but for our purposes this is enough. p is for "paragraph" (the text between <p> and </p> will be separated by some space before the next element is displayed in the final HTML page rendering), h1 means "heading level 1", and will be rendered by default in bold with a bigger char size than any other text element, etc.

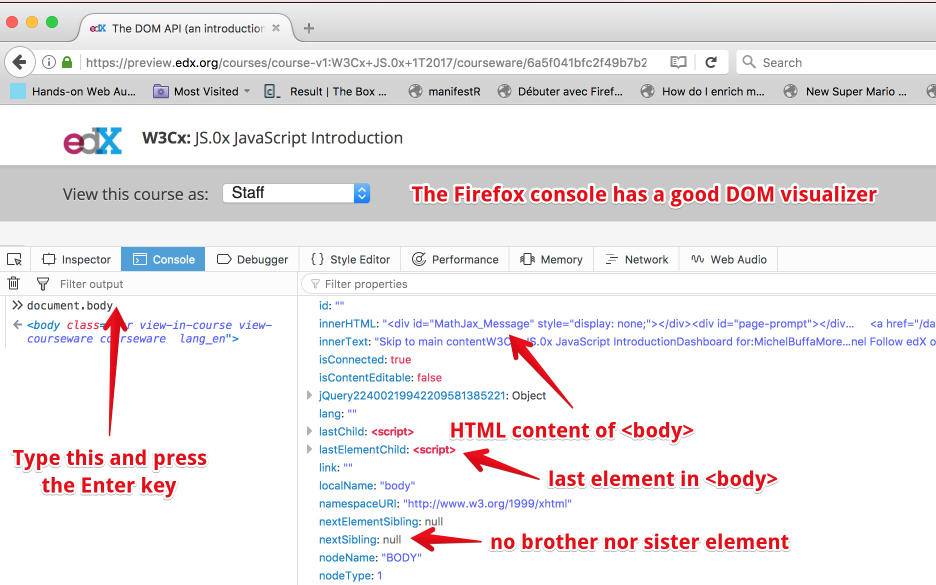
#### There are different types of nodes in the DOM

There are different types of nodes, but don't worry - the most useful ones are highlighted in bold.

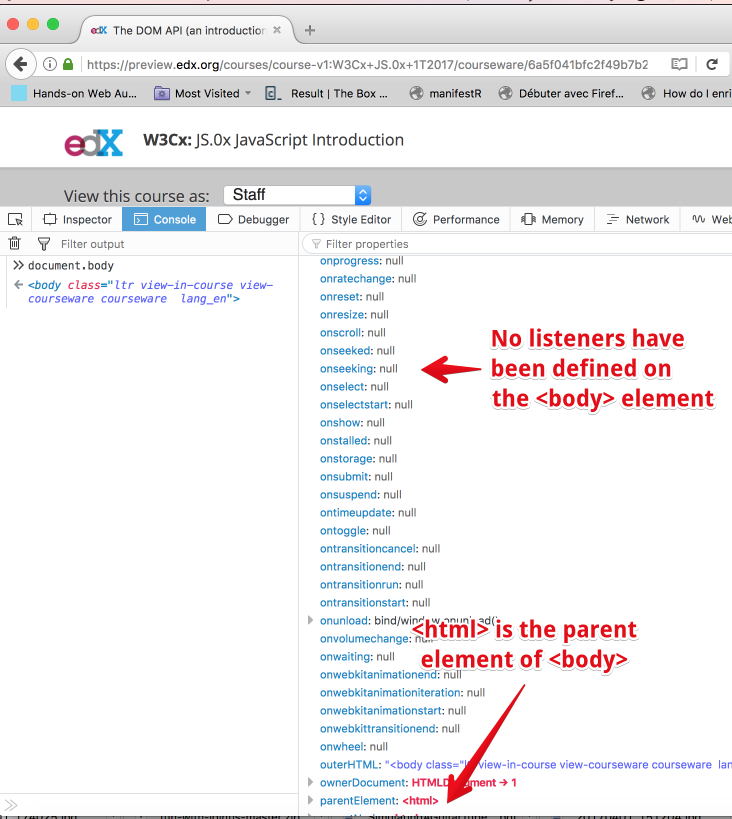
* **Element (example: <ul></ul>)**
* **Text (example: <p>the text within the element p is a node of type text</p>)**
* Document, DocumentFragment, DocumentType (example: <!doctype html> for html5), Comment (example: <!-- left column -->), ProcessingInstruction (example: <?php echo $name ?>)

#### Exploring the DOM with the devtool console

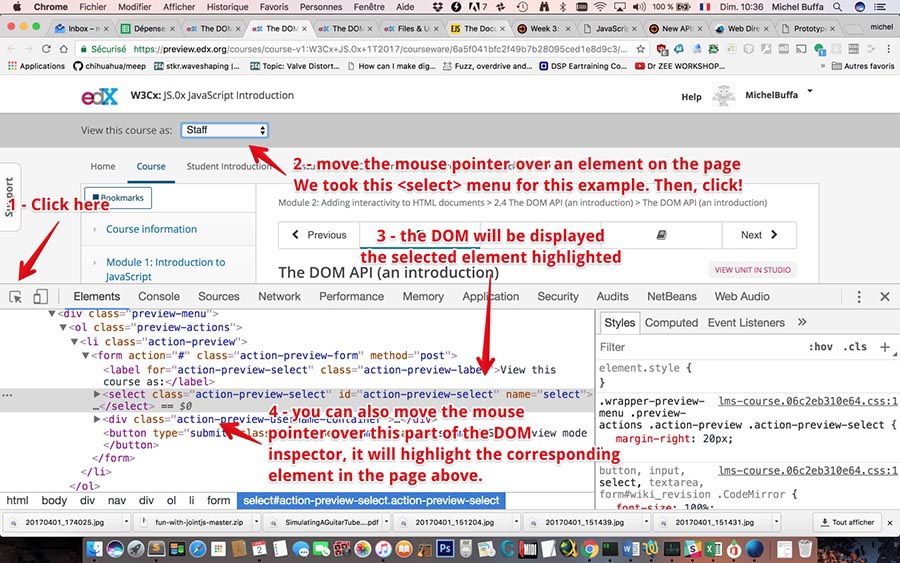
You can explore the DOM with the devtool console. This time we used Firefox for exploring the DOM, as it proposes a good structured view of the DOM and of its properties/methods:



If you scroll down the right panel of the devtool console, as in the above screenshot, you will be able to look at all the properties, all the methods, all the event listeners:



You can also use the "DOM inspector" to locate a particular element with the mouse: click the target icon and click on  the element on the page that you want to inspect, this time with Google Chrome, but you will find this option in all modern browsers' devtool consoles:



* 1. **A warning about the DOM API**

### **A warning about the DOM API**

The DOM and the DOM API can be cumbersome and complicated. There are many methods and properties for manipulating the DOM tree, that are not "very JavaScript". There are historical reasons for this: the DOM wasn’t designed exclusively for JavaScript. Rather, it tries to define a language-neutral interface that can be used in other systems as well — not just HTML but also XML, which is a generic data format with an HTML-like syntax.

HTML5 made some additions that are not in the DOM API but which greatly help the JavaScript programmer (we'll see this in a minute with the "selector API", for example).

So we've decided to focus on only 20% of the DOM API and on the selector API (for selecting elements in the DOM). These are the most useful parts and it will give you enough knowledge to solve nearly every problem where you need to manipulate the DOM.

* 1. **Accessing HTML elements**

**[ -- Live coding video: accessing HTML elements -- ]**

#### 1 - With the selector API (recommended)

Extract from [HTML5 selectors API – It’s like a Swiss Army Knife for the DOM](http://www.webdirections.org/blog/html5-selectors-api-its-like-a-swiss-army-knife-for-the-dom/) : "One of the many reasons for the success of JavaScript libraries like [*jQuery*](https://jquery.org/) and [*Prototype*](http://prototypejs.org/), on top of their easing the pain of cross-browser development was how they made working with the DOM far less painful than it had previously been, and indeed how it was with the standard DOM. Being able to use arbitrary CSS selector notation to get matching elements from a document made the standard DOM methods seem antiquated, or at the very least, far too much like hard work.

Luckily, the standards and browser developers took notice. The W3C developed the Selectors API, a way of easily accessing elements in the DOM using standard CSS selector concepts, and browser developers have baked these into all modern browsers, way back to IE8."

##### **The querySelector(CSSSelector) and querySelectorAll(CSSSelector) methods**

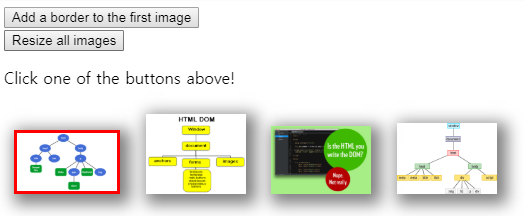
Ah... these methods owe a lot to [jQuery](https://jquery.org/)! They introduce a way to use CSS selectors (including CSS3 selectors) for requesting the DOM, like jQuery introduced ages ago.

Any CSS  selector can be passed as a parameter for these methods.

* While **querySelector(selector)** will return the first element in the DOM that matches the selector (and you will be able to work with it directly),
* **querySelectorAll(selector) returns a collection of HTML elements corresponding to all elements matching the selector**. To process the results, it will be necessary to loop over each of the elements in the collection.

##### **Typical use:**

Looking for an element in the whole document (the whole HTML page): call the querySelector method (or querySelectorAll) on the document object, that corresponds to the whole DOM tree of your web page:



**Source code from the above example:**

**HTML part**: we have two buttons that will call a JavaScript function (lines 2 and 6) where we will manipulate the DOM), and we have four images, the first one with an id equal to "img1" (lines 11, 14, 16 and 18).

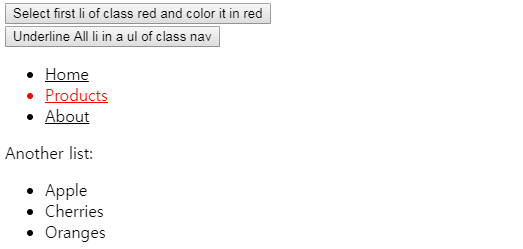
1. ...
2. <button onclick="addBorderToFirstImage();">
3. Add a border to the first image
4. </button>
5. <br>
6. <button onclick="resizeAllImages();">
7. Resize all images
8. </button>
9. <br>
10. <p>Click one of the buttons above!</p>
11. <img src="http://i.imgur.com/Ntvj5rq.png"
12. id="img1"
13. width=200>
14. <img src="http://i.imgur.com/yiU59oi.gif"
15. width=200>
16. <img src="http://i.imgur.com/6FstYbc.jpg"
17. width=200>
18. <img src="http://i.imgur.com/L97CyS4.png"
19. width=200>
20. ...

**JavaScript part**: the init function is executed as soon as the page is loaded (and the DOM is ready), in this function we add a shadow and margins to all images (lines 3-21). The two other functions are called when one of the HTML buttons is clicked (line 23 and line 31).

1. **window.onload = init; // run init once the page is loaded**
3. **function init()** {
4. // we're sure that the DOM is ready
5. // before querying it
6. // **this function runs once the page is loaded**
7. // add a shadow to all images
8. // **select all images**
9. var listImages =**document.querySelectorAll("img");**
11. // change all their width to 100px
12. listImages.forEach(function(img) {
13. // img = current image
14. // add a shadow 5px left, 5 pixel down, 15px blur, 5px spread
15. // grey
16. img.style.boxShadow = "5px 5px 15px 5px grey";
17. // add a margin 10px on each side
18. img.style.margin = "10px";
19. });
20. }
22. **function addBorderToFirstImage()** {
23. // select the first image with id = img1
24. var img1 =**document.querySelector('#img1');**
26. // Add a red border, 3px wide
27. img1.style.border = '3px solid red';
28. }
30. **function resizeAllImages()** {
31. // select all images
32. var listImages = document.querySelectorAll("img");
34. // change all their width to 100px
35. listImages.forEach(function(img) {
36. // img = current image, we resize it by changing its
37. // width attribute
38. img.width = 100;
39. });
40. }

##### **Miscellanous examples of use of querySelector(CSSSelector) and querySelectorAll(CSSselector)**

Here are some other examples that use more complicated CSS selectors. If you are not familiar with their syntax, we recommend that you follow the CSS basics, and HTML5 and CSS fundamentals courses from [W3Cx](https://www.edx.org/school/w3cx).

**Example 1: get all <li> directly in a <ul> of class nav:** 

Source code extracts:

HTML:

1. <button onclick="**firstLiClassRedInUl()**;">Select first li of class red and color it in red</button>
2. <br>
3. <button onclick="**allLisInUlOfClassNav()**;">Underline All li in a ul of class nav</button>
4. <ul **class="nav"**>
5. <li>Home</li>
6. <li **class="red"**>Products</li>
7. <li>About</li>
8. </ul>
9. Another list:
10. <ul>
11. <li>Apple</li>
12. <li **class="red"**>Cherries</li>
13. <li>Oranges</li>
14. </ul>

JavaScript code:

1. **function firstLiClassRedInUl()** {
2. // first li of class="red" in a ul
3. var elm =**document.querySelector("ul li.red");**
4. elm.style.color = 'red';
5. }
7. **function allLisInUlOfClassNav()** {
8. // get all li directly in a ul of class nav
9. var list =**document.querySelectorAll("ul.nav > li");**
10. list.forEach(function(elm) {
11. elm.style.textDecoration = "underline";
12. })
13. }

**Example 2: display all checked <input type="checkbox"> elements located inside an element of a given id:**

****

Extract from the source code:

HTML**:**

1. <button onclick="**displayListOfCheckedItems();**">
2. Show Checked items
3. </button>
4. <br>
5. <ul id="fruits">
6. <li>
7. <input type="checkbox" name="fruit" value="apples">
8. Apples
9. </li>
10. <li>
11. <input type="checkbox" name="fruit" value="oranges">
12. Oranges
13. </li>
14. <li>
15. <input type="checkbox" name="fruit" value="bananas">
16. Bananas
17. </li>
18. <li>
19. <input type="checkbox" name="fruit" value="grapes">
20. Grapes
21. </li>
22. </ul>

JavaScript code: we select all elements of type input that have an attribute checked equal to true, and located inside an element whose id is "fruits". Notice the use of document.querySelectorAll, for selecting more than one element (line 6), then, we iterate on the list (line 8) and concatenate to the string variable listOfSelectedValues the value of each element (located in its value attribute). This is done in line 9.

Lines 9-12 use the parentNode property of the selected nodes in order to change the color of the <li> (parents of <input> elements selected) in red. In the DOM tree, we selected input elements that are each a child of a <li> element. The text displayed: "Apples", "Oranges" etc. belong to the <li> element. In order to access it from the <input> child we selected, we use elm.parentNode.

Finally, at the end of the document, line 14 adds a message followed by this list:

1. function displayListOfCheckedItems() {
2. // all inputs that have been checked
3. var listOfSelectedValues="";
4. var list = document.querySelectorAll("#fruits input:checked");
5. list.forEach(function(elm) {
6. listOfSelectedValues += elm.value + " ";
7. // Put the li in red.
8. // the li is the parent of the current input elem stored
9. // in the elm variable
10. elm.parentNode.style.color = 'green';
11. });
12. document.body.append("You selected: " + listOfSelectedValues);
13. }

**Example 3: change the background of all paragraphs <p> in an element of a given id:**

HTML code:

1. <button onclick="**changeBackGroundOfPs('firstDiv');**">Change backgrounds of p under a given element known by id</button>
2. <br>
3. <div **id="firstDiv"**>
4. <p>First paragraph.</p>
5. <p>Second paragraph.</p>
6. </div>

JavaScript code: we build a CSS selector using the id passed as a parameter. In this example, the id is 'firstDiv', the id of the div at line 3 in the above code.

So, the variable CSS selector at line 2 in the JavaScript code below will have a value equal to "#firstDiv p", that means: select all <p> under an element whose id is "firstDiv". The paragraphs variable is a list that contains the paragraphs selected. Then we iterate on this list (this time using a for loop, which is an alternative method to using the forEach method used in previous examples) (line 5-7), and we change the background of all selected paragraphs (line 6).

1. function changeBackGroundOfPs(id) {
2. var paragraphs = document.querySelectorAll("#" + id + " p");
4. // Another way to iterate on all elements in a collection
5. for (var i = 0; i < paragraphs.length; i++ ) {
6. paragraphs[i].style.backgroundColor = "lightGreen";
7. }
8. }

**Other examples that use more complex selectors**:

1. // all elements li in ul elements in an element of id=nav
2. var el = document.querySelector('#nav ul li');
3. // all li in a ul, but only even elements
4. var els = document.querySelectorAll('ul li:nth-child(even)');
5. // all td directly in tr in a form of class test
6. var els = document.querySelectorAll('form.test > tr > td');
7. // all paragraphs of class warning or error
8. querySelectorAll("p.warning, p.error");
9. // first element of id=foo or id=bar
10. querySelector("#foo, #bar");
11. // first p in a div
12. var div = document.getElementById("bar");
13. var p = div.querySelector("p");

#### 2 - With the DOM API (old fashioned)

These methods are from the DOM API and can all be replaced by the querySelector and querySelectorAll methods that we've discussed. They are still used in many JavaScript applications, and are very simple to understand.

From the document we can access the elements composing our web page in a few ways:

* **document.getElementById(identifier)** returns the element which has the id “identifier”.
* This is equivalent to document.querySelector("#identifier'); (just add a # before the id when using a CSS selector).   
    
  Example: var elm = document.getElementById('myDiv'); is equivalent to document.querySelector('#myDiv');
* **document.getElementsByTagName(tagName)**returns a list of elements which are named “tagName”.
* This is equivalent to document.querySelectorAll(tagName);
* Example: var list = document.getElementByTagName('img'); is equivalent to document.querySelector('img');
* **document.getElementsByClassName(className)** returns a list of elements which have the class “className”.

This is equivalent to document.querySelectorAll('.className');

Example: var list = document.getElementByClassName('important'); is equivalent to document.querySelector('.important'); (just add a '.' before the class name when using a CSS selector).

Notice that identifier, tagName and className must be of type String.

* 1. **Changing the style of selected HTML elements**

#### The style attribute: how to modify an HTML element's CSS properties from JavaScript

The most common way to modify the CSS style of one of several elements you selected using the DOM or Selector API, is to use the style attribute.

Typical use:

1. // select the paragraph with id = "paragraph1"
2. var p = document.querySelector('#paragraph1');
4. // change its color
5. p.style.color = 'red';

Warning: with the style attribute, you can modify (or read) any CSS property, but be careful: the syntax changes a little due to the fact that in JavaScript the "-" is a math operator, while in CSS it is used to separate properties made of multiple words, such as background-color.

When using such properties from JavaScript, the rule is simple:

1. Remove the "-" sign,
2. Capitalize the word after the "-" sign!

Simple, isn't it?

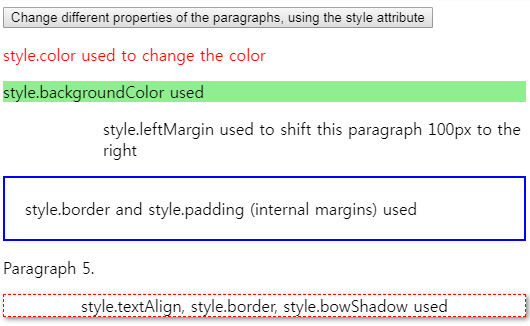
Examples:

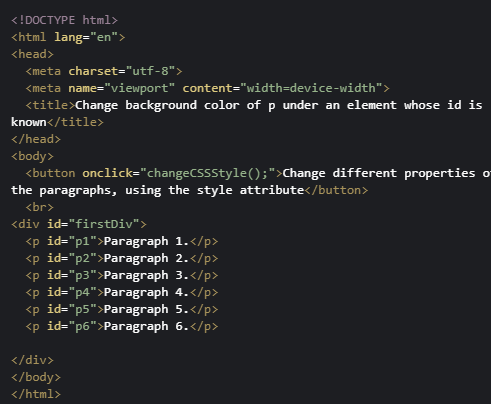
* text-align becomes style.textAlign
* margin-left becomes style.marginLeft
* etc.

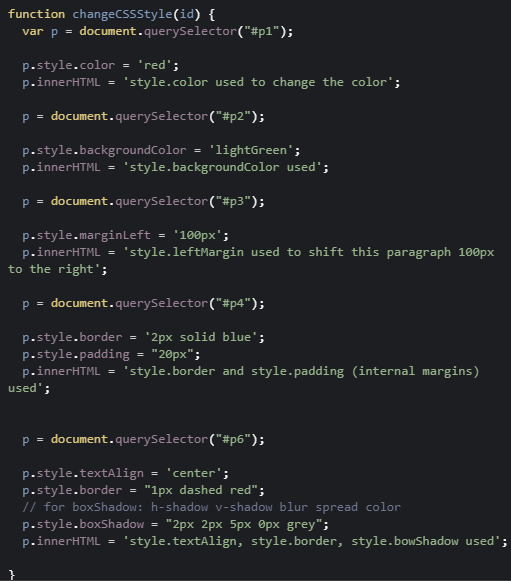
The most useful CSS properties (we do recommend that you follow the W3Cx courses CSS basics, CSS and HTML5 fundamentals from W3Cx to learn more about CSS):

* color: changing the color of the text content of selected element(s),
* background-color: same but this time the background color,
* margin and padding properties (external and internal margins), including their variants: margin-left, margin-top, margin-right, margin-bottom, also padding-left, etc.
* border and border-radius: change the border, type (plain, dashed), color, thickness, rounded corners etc.
* box-shadow to add shadows to selected elements,
* font, font-style: font characters and style (italic, bold, plain)
* text-align (centered, etc.)

Here are some examples:



****

****

#### Using the ClassList interface to change more than one CSS property simultaneously

External resources:

* [The W3C specification about the classList DOM interface](http://www.w3.org/TR/dom/#dom-element-classlist)
* [An article from the Mozilla Developer's web site](https://hacks.mozilla.org/2010/01/classlist-in-firefox-3-6/)

Until now, to manipulate CSS classes of an HTML element was a bit complex, both for verifying the presence of a class name in an element, and for adding or removing classes associated with a given element.

The ClassList interface simplifies it all by acting as a container object and by providing a set of methods to manipulate its content.

The classList property applies to an HTML element, and returns a collection of class names:

1. var elem= document.querySelector("#id1");
3. var allClasses = elem.**classList**;

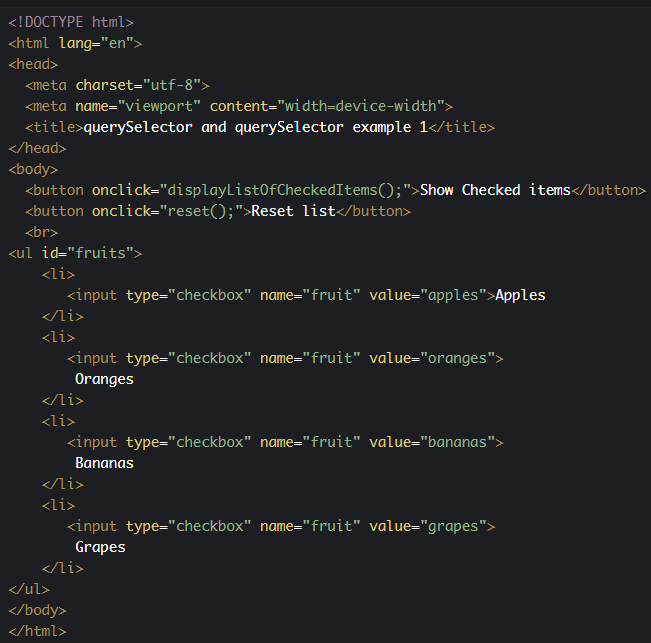
##### **The classList API**

The list of methods usable on a classList object are add(), remove(), toggle() and contains().

1. // By default, start without a class in the div: <div class=""/>
2. // Set "foo" as the class by adding it to the classList
3. div.classList.add('foo'); // now <div class="foo"/>
4. // Check that the classList contains the class "foo"
5. div.classList.contains('foo'); // returns true
6. // Remove the class "foo" from the list
7. div.classList.remove('foo'); // now <div class=""/>
8. // Check if classList contains the class "foo"
9. div.classList.contains('foo'); // returns false: "foo" is gone
10. // Check if class contains the class "foo",
11. // If it does, "foo" is removed, if it doesn't, it's added
12. div.classList.toggle('foo'); // class set to <div class="foo"/>
13. div.classList.toggle('foo'); // class set to <div class=""/>

Another example: add and remove multiple CSS properties in a list of checkboxes.

****

****

This is a variation of an example from a previous section. This time, when the <input type="checkbox"> elements have been checked, in order to give the parent <li> a background color, a border, and to change the text color, we use a CSS class named "checked":

CSS code:

1. .checked {
2. border: 2px dashed #000;
3. background-color: green;
4. color:yellow;
5. }

... and the classList.add(CSS\_class) and classList.remove(CSS\_class) methods on the <li> elements:

JavaScript code:

1. function displayListOfCheckedItems() {
2. // all inputs that have been checked
3. var listOfSelectedValues="";
4. var list = document.querySelectorAll("#fruits input:checked");
5. list.forEach(function(elm) {
6. listOfSelectedValues += elm.value + " ";
7. **// get the li parent of the current selected input**
8. **var liParent = elm.parentNode;**
9. **// add the CSS class .checked**
10. **liParent.classList.add("checked");**
11. });
12. document.body.append("You selected: " + listOfSelectedValues);
13. }
15. function reset() {
16. var list = document.querySelectorAll("#fruits input");
17. list.forEach(function(elm) {
18. // uncheck
19. elm.checked = false;
20. **// remove CSS decoration**
21. **var liParent = elm.parentNode;**
22. **liParent.classList.remove("checked");**
23. });
24. }
    1. **Changing the content of selected HTML elements**

### **Modifying selected HTML elements**

We've already seen many examples (even during week 1) in which we selected one or more elements, and modified their content. Let's summarise all the methods we've seen, and perhaps introduce a few new things...

#### Properties that can be used to change the value of selected DOM node

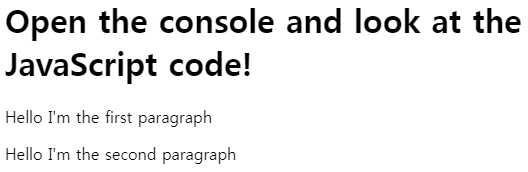
##### **Using the innerHTML property**

This property is useful when you want to change all the children of a given element. It can be used to modify the text content of an element, or to insert a whole set of HTML elements inside another one.  
  
Typical use:

1. var elem = document.querySelector('#myElem');
3. elem.innerHTML = 'Hello '; // replace content by Hello
5. elem.innerHTML += '<b>Michel Buffa</b>', // append at the end
6. // Michel Buffa in bold
8. elem.innerHTML = 'Welcome' + elem.innerHTML; // insert Welcome
9. // at the beginning
11. elem.innerHTML = ''; // empty the elem

##### **Using the textContent property**

It's also possible, with selected nodes/elements that contain text, to use the textContent property to read the text content or to modify it. There are subtle differences that can be seen in the above example

****

Extract from the HTML code:

1. <p id="first">first paragraph</p>
2. <p id="second"><em>second</em> paragraph</p>

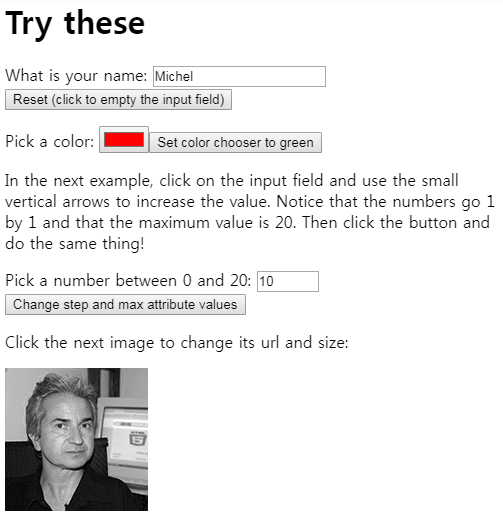
JavaScript code: the comments after lines that start with console.log correspond to what is printed in the devtool debug console. Notice the difference between the textNode value and the innerHTML property values at lines 13-14: while textContent returns only the text inside the second paragraph, innerHTML also returns the <em>...</em> that surrounds it. However, when we modify the textContent value, it also replaces the text decoration (the <em> is removed), this is done at lines 16-20.

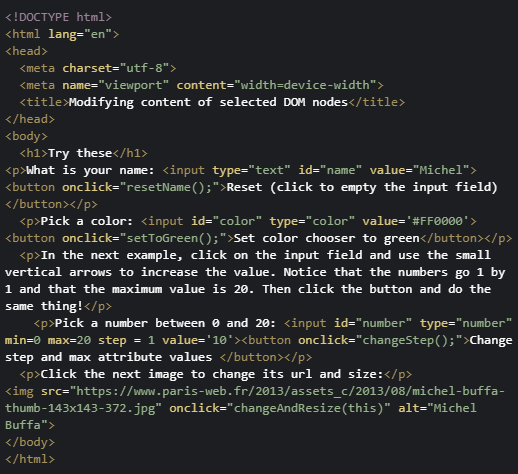
1. window.onload = init;
3. function init() {
4. // DOM is ready
5. var firstP = document.querySelector("#first");
6. console.log(firstP.textContent); // "first paragraph"
7. console.log(firstP.innerHTML);   // "first paragraph"
9. firstP.textContent = "Hello I'm the first paragraph";
10. console.log(firstP.textContent); // "Hello I'm the first paragraph"
12. var secondP = document.querySelector("#second");
13. console.log(secondP.textContent); // "second paragraph"
14. console.log(secondP.innerHTML);   // "<em>second</em> paragraph"
15. secondP.textContent = "Hello I'm the second paragraph";
16. console.log(secondP.textContent); // "Hello I'm the second
17. // paragraph"
18. console.log(secondP.innerHTML);   // "Hello I'm the second
19. // paragraph"
20. }

#### Changing the attributes of selected elements

It's very common to modify the attributes of selected elements: the width of an image, CSS style with the style attribute, value of an input field, etc.

This example shows some of the things we can do:

****

****

****

* 1. **Adding new elements to the DOM**

The DOM API comes with a set of methods you can use on DOM elements.

In general, to add new nodes to the DOM we follow these steps:

1. Create a new element by calling the createElement() method, using a syntax like:
2. var elm = document.createElement(name\_of\_the\_element).

Examples:

1. var li = document.createElement('li');
2. var img = document.createElement('img'); etc.
3. Set some attributes / values  / styles for this element.  
     
   Examples:
4. li.innerHTML = '<b>This is a new list item in bold!</b>'; // can add HTML in it
5. li.textContent = 'Another new list item';
6. li.style.color = 'green'; // green text
7. img.src = "http://..../myImage.jpg"; // url of the image
8. img.width = 200;
9. Add the newly created element to another element in the DOM, using append(), appendChild(), insertBefore() or the innerHTML property  
     
   Examples:
10. var ul = document.querySelector('#myList');
11. ul.append(li); // insert at the end, appendChild() could also be used (old)
12. ul.prepend(li); // insert at the beginning
13. ul.insertBefore(li, another\_element\_child\_of\_ul);// insert in the middle
14. document.body.append(img); // adds the image at the end of the document

#### A warning about append vs appendChild, prepend, etc...

**If the examples do not run on your browser (Edge, IE, old browsers), read this!**

The DOM specification recently added some jQuery like methods that are similar to the ones proposed by the [jQuery](https://jquery.org/) library (that was designed a long time ago to simplify DOM manipulations).

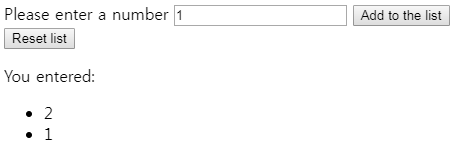
For a long time, developers used document.appendChild to add an element to the DOM. Now you can use document.append, which is shorter, along with some other methods such as document.prepend etc. All these methods are supported by recent browsers except Microsoft Edge (but support is coming soon). See [this table for compatibility](http://caniuse.com/#feat=dom-manip-convenience).

**In the course, we recommend that you use document.append, but if you are looking for maximum compatibility, you can either use document.appendChild instead of document.append (just search and replace all occurrences of document.append with document.appendChild in the examples), or add to your JavaScript code**[**this polyfill**](https://developer.mozilla.org/fr/docs/Web/API/ParentNode/append#Polyfill)**.**

Just copy and paste the 10 lines of JavaScript from the polyfill to your code and append will work on browsers that do not yet implement it.

Here is [an example in JsBin that uses all these new methods, and that includes the polyfill](http://jsbin.com/sodeheh/edit?html,js,output) for append and prepend at the end of the JavaScript code.

#### Example1: use of the createElement(), append() methods and of the textContent attribute:

****

HTML code extract: we use an <input type="number"> for entering a number (*line 2*). Then if one clicks on the "Add to the list" button, the add() JavaScript function is called (*line 3*), this will add the typed number to the empty list at*line 7*. If one presses the "reset" button, it will empty this same list by calling the reset() JavaScript function.

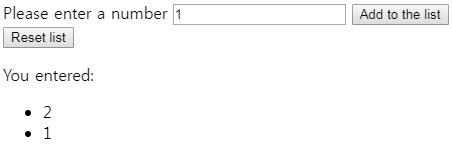
1. <label for="newNumber">Please enter a number</label>
2. <input type="number" id="newNumber" value=0>
3. <button onclick="**add();**">Add to the list</button>
4. <br>
5. <button onclick="**reset();**">Reset list</button>
7. <p>You entered:</p>
8. <ul id="numbers"></ul>

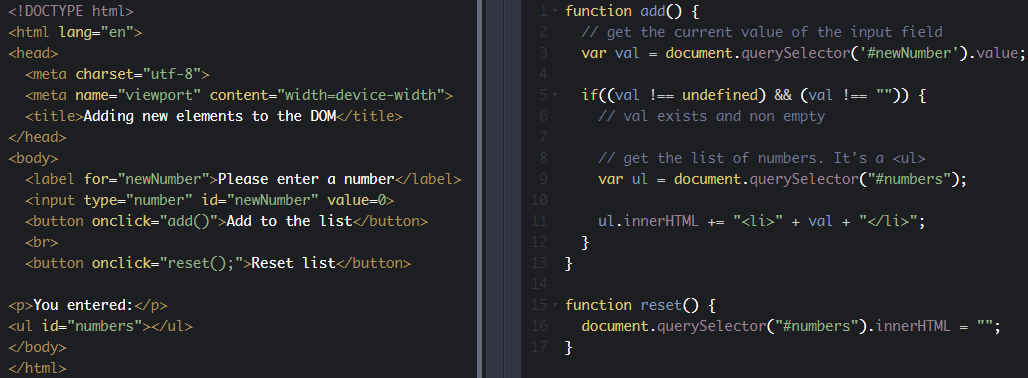
JavaScript code extract: notice at *line 25* the use of the innerHTML property for resetting the content of the <ul> list. innerHTML corresponds to all the sub DOM contained inside the <ul>...</ul>. InnerHTML can be used for adding/deleting/modifying a DOM node's content.

1. function add() {
2. // get the current value of the input field
3. var val = document.querySelector('#newNumber').value;
4. if((val !== undefined) && (val !== "")) {
5. // val exists and non empty
6. // get the list of numbers. It's a <ul>
7. var ul = document.querySelector("#numbers");
8. // add it to the list as a new <li>
9. var newNumber =**document.createElement("li");**
10. **newNumber.textContent = val;**
11. // or newNumber.innerHTML = val
12. **ul.append(newNumber);**
13. }
14. }
16. function reset() {
17. // get the list of numbers. It's a <ul>
18. var ul = document.querySelector("#numbers");
19. // reset it: no children
20. **ul.innerHTML = "";**
21. }

#### Example 2: using the innerHTML property to add new elements

This is the same example, but in an abbreviated form, using the innerHTML property:

****

****

* 1. **Moving HTML elements in the DOM**

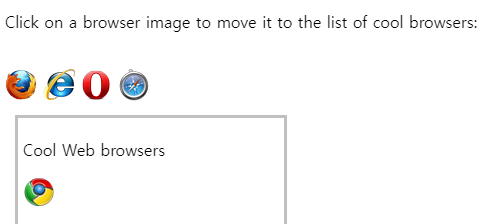
The append(), appendChild() methods normally adds  a new element to an existing one, as shown in this example:

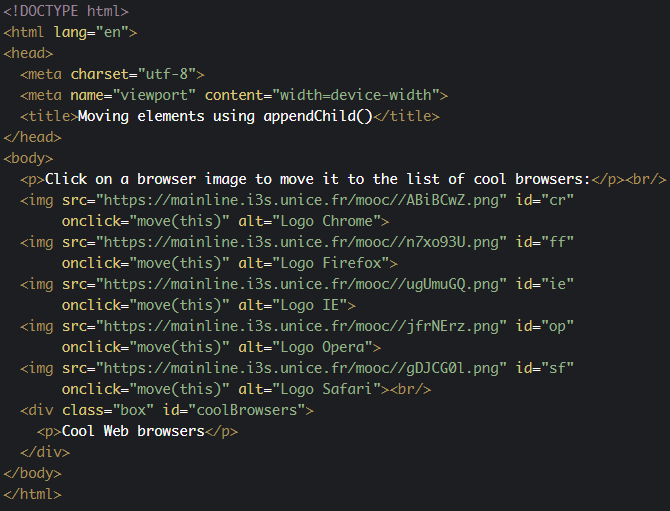
1. var li = createElement('li');
2. ul.append(li); // adds the new li to the ul element

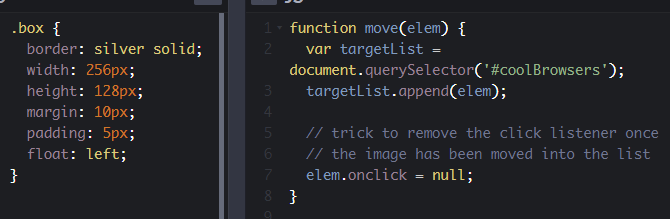
One interesting thing to know is that if we do not create the new element, but rather get it from somewhere else in the document, it is then removed from its parents and added to the new parent.

In other words: it moves from its original location to become a child of the targetElem.

#### Let's see a very simple example:

****

****

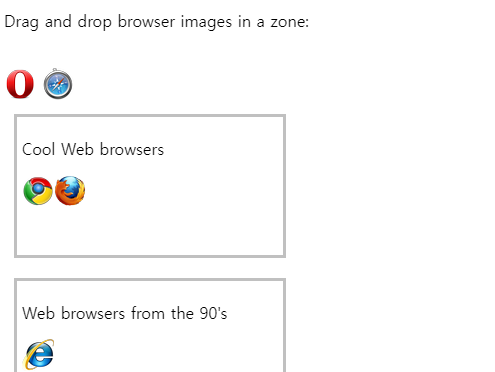
****

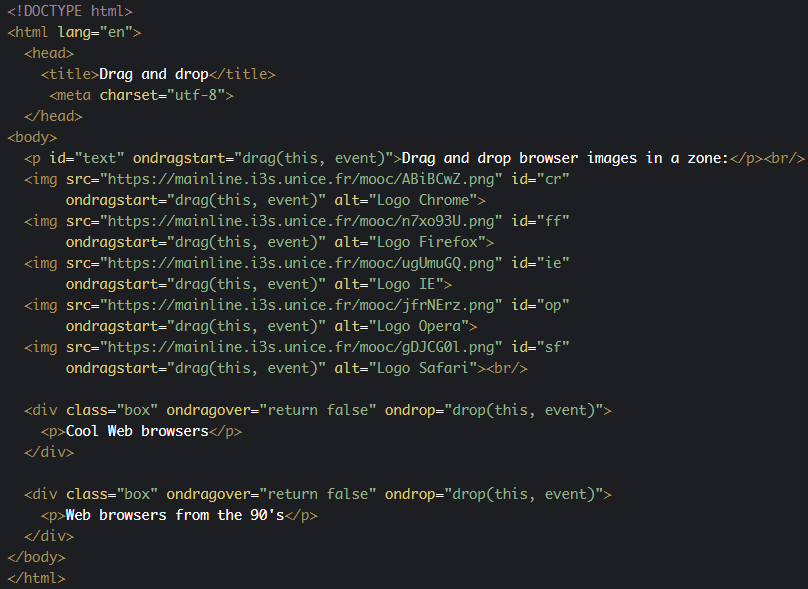
#### Another, more significant example, that also uses drag'n'drop

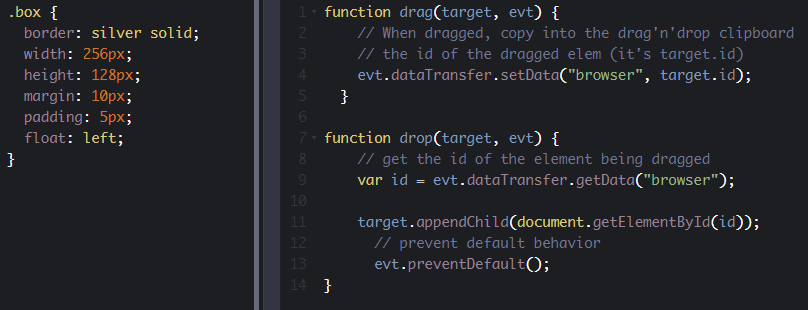
Notice that this example comes from the HTML5 advanced course. Our plan here is not to explain drag'n'drop in detail, but to show how append() can be used to move an element.

In this example, when a user starts to drag an element, the drag() JavaScript function is called. In this function we use the drag'n'drop clipboard to store the id of the image that is being dragged.

When the image is dropped, the drop() method is called. As the drop event listener is declared on the two divs (on the left and the right), we just call append() on the target div element, and this will add the dragged image to the div, while removing it from its original location.



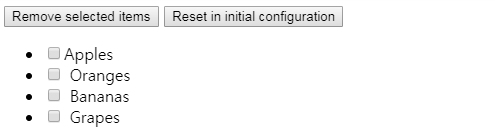
****

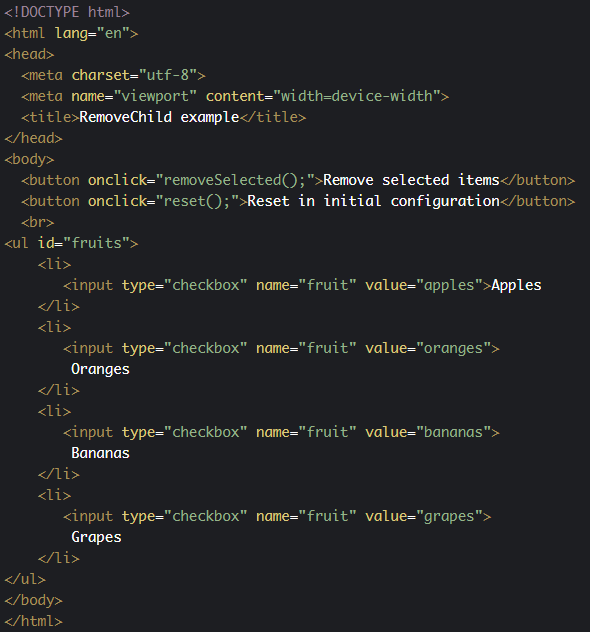
****

* 1. **Removing elements from the DOM**

#### Removing elements using the removeChild() method

Let's take an example that we've already encountered. This time, you will check the elements you want to remove from the list!

****

****

****

JavaScript code extract: we need to get the <ul> that contains all the <li><input type="checkbox"></li> elements (line 3). This is the element we will use for calling removeChild(...). The loop on the checked element (lines 5-12) iterates on a list of checked input elements. In order to make both the text (Apples, Oranges, etc.) AND the checkbox disappear, we need to access the different <li> elements that contain the selected checkboxes. This is done in line 10. Then, we can call ul.removeChild(li) on the <ul> for removing the <li> that contains the selected element (line 11).

1. function removeSelected() {
2. var list = document.querySelectorAll("#fruits input:checked");
3. **var ul = document.querySelector("#fruits");**
4. list.forEach(function(elm) {
5. // elm is an <input type="checkbox">, its parent is a li
6. // we want to remove from the <ul> list
7. // when we remove the <li>, the <input> will also
8. // be removed, as it's a child of the <li>
9. **var li = elm.parentNode;**
10. **ul.removeChild(li);**
11. });
12. }

#### Remove all children of an element using the innerHTML property

In the same example, if you look at the reset() JavaScript function, we use the ul's innerHTML property both for emptying the list (lines 3-4) and for appending to it all the initial HTML code (lines 6-17):

1. function reset() {
2. var ul = document.querySelector("#fruits");
3. **// Empty the <ul>**
4. **ul.innerHTML = "";**
5. // Adds each list item to the <ul> using innerHTML += ...
6. **ul.innerHTML +=** "<li><input type='checkbox' name='fruit'    
                      value='apples'>Apples</li>";
8. **ul.innerHTML +=** "<input type='checkbox' name='fruit'
9. value='oranges'>Oranges</li><br>";
11. **ul.innerHTML +=** "<input type='checkbox' name='fruit'
12. value='bananas'>Bananas</li><br>";
14. **ul.innerHTML +=** "<input type='checkbox' name='fruit'
15. value='grapes'>Grapes</li>";
16. }
    1. **Discussion topics**

Here is the discussion forum for this part of the course. Please either post your comments/observations/questions or share your creations.

See below for suggested topics of discussion.

#### Suggested topics

* Did you know there were different keyboard layouts?
* Did you know the best practices in order to make a Web application compatible with different keyboard layouts?
* Did you know there were different properties for getting the mouse coordinates?
* Did you know the method we proposed for getting the mouse coordinate relative to the element the mouse is being moved on?
* Sometimes, detecting key events on a canvas HTML element is tricky. Do not forget to visit the [HTML5 Coding essentials and Best practices MOOC at W3Cx](https://www.edx.org/course/html5-coding-essentials-w3cx-html5-1x-2), as we cover these in details in the section about the HTML5 canvas element.