What you will learn in Module 2

Azzeddine
RIGAT

- **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

Boolean values and logical operators

Azzeddine RIGAT

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.

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

```
var a = true, b = false, c = 'true';

// c is not a boolean but a string
```

- **Undefined and null values**
  - Undefined:undefined is returned when a variable has not been assigned

```
var foo;
foo; //Output: undefined
typeof foo; //Output: undefined
if (foo === undefined) //Test whether a variable has a value
console.log('The variable foo has no value and is undefined');
```

Boolean values and logical operators

Azzeddine RIGAT

- **Undefined and null values**
  - 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":

```
bar; //Output: ReferenceError
typeof bar; //'undefined'
```

Boolean values and logical operators

Azzeddine RIGAT

- 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.**

Boolean values and logical operators

Azzeddine
RIGAT

- Implicit conversions of non boolean values in expressions:

```
var b = "one";
!b;   //OUtput: false, implicit conversion of "one" to a boolean value
var b = "one"; // implicit conversion of "one" to a boolean value
!!b; //Output: true
```

  - All the following values are evaluated as false :
    - **false, undefined, null, 0, NaN, the empty string ''**
  - Everything else is evaluated as true!
    - Tips
      - `var boo = 'hello' || 'world'; // boo is equal to 'hello' that is 'true`
      - `var boo = 'hello' && 'world'; // boo is equal to 'world' that is 'true'`

Boolean values and logical operators

Azzeddine
RIGAT

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.

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.

```
var boo2 = (0/0) || 43.2 ;
```

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

Boolean values and logical operators

Azzeddine
RIGAT

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**.

Boolean values and logical operators

Azzeddine RIGAT

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.

```
1 == 1 : //true
1 == 2 : //false
/* Here, the interpreter will try to convert the string '1'
into a number before doing the comparison */
1 == '1': //true :
//with strict equal, no conversion:
1 === 1: //true
1 === '1': //false
```

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

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

Boolean values and logical operators

Azzeddine
RIGAT

NaN has this special property :

```
NaN == NaN; // false
NaN === NaN; // 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.

```
isNaN(NaN); // true
isNaN(0/0); // true
isNaN(12); // false
isNaN('foo'); // true
```

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

Boolean values and logical operators

Azzeddine RIGAT

More example on NaN:

```
var num =0/0;
//First version:
if(isNaN(num)){
  num = 0;
}
//Second version: shortened version with the conditional operator (ternary operator)
var num = isNaN(num) ? 0 : num
var num = num ? num : 0
//Third version: version with logical operator (implicit conversion)
var num = num || 0;

num; //returns 0 in this three cases
```

Conditional statements: if...then...else, switch

Azzeddine
RIGAT

What are statements?

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).

```
var myVar = 'hello ' + 'world';
```

The block statement:

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

```
{
var i = 0;
 var result = false;
 console.log('i = ' + i);
}
```

Conditional statements: if...then...else, switch

Azzeddine RIGAT

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

Conditional statements: **check exmaple.II.2.2.0**, (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.

Syntax:

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

Example:

```
var num = 10;
if (num > 10) {
   num = 20;
} else {
   num = 0;
}
// num equals 0
```

Conditional statements: if...then...else, switch

Azzeddine RIGAT

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

Syntax:

var = condition ? (Result if your **condition is true**) **:** (Result if your **condition is false**)

Let's look at this code example:

```
var max;
var min = 2;
if (min < 10) {
   max = min + 10;
} else {
   max = min;
}
```

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

The equivalent code is:

```
var max;
var min;
max = (min < 10) ? min+10 : min;
```

Conditional statements: if...then...else, switch

Azzeddine RIGAT

The switch statement; In order to avoid having a series of if and else, it is possible to use a switch statement.

The syntax of the switch statement is:

```
switch (expression) {
   case value1:
       statement
       break;
   ...

   case valueN:
       statement
       Break;

   default:      // if no case tested true
       statement
       break;
}
```

Example: check exmaple.II.2.2.0(line 38)

```
var gear = '';
switch (cloudColor) {
   case 'green':
       gear = 'spacesuit';
       break;

   case 'white':
       gear = 'jacket';
       break;
   default:
       gear = 'watch';
       break; // useless if in the last case
} // end of the switch statement
```

Loop statements, The while statement

Azzeddine
RIGAT

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 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.

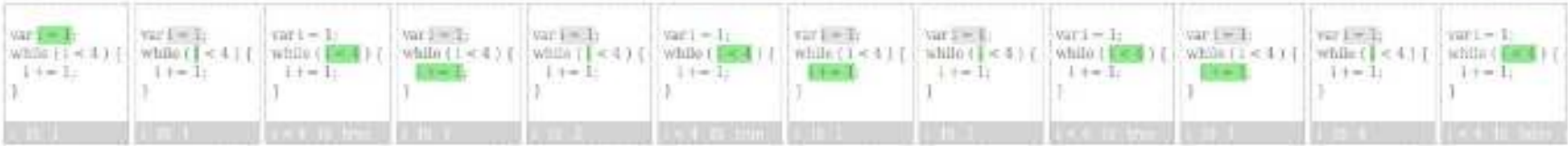
The while statement: With a while statement, a block of code is executed repeatedly while the specified condition is satisfied (evaluates to true).

Syntax:

```
while (condition) statement
```

Example:

```
var i = 1;
while ( i < 4 ) {
   i += 1;
}
```



Loop statements, The do-while statement

Azzeddine
RIGAT

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

Syntax:

```
do statement while (codition)
```

Example:

```
var i = 0;
do {
   console.log('i = ' + i);
   i++;
} while(i < 20);
console.log('Value of i after the do-while
statement: ' + i);
```

Loop statements, The do-while statement

Azzeddine
RIGAT

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.

Loop statements, The for statement

Azzeddine
RIGAT

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

Syntax:

```
for (initialization; condition; incrementation) statement
```

Example:

```
for (var i = 0; i < 4; i++) {
   i = i +1 ;
}
```

Loop statements, The for statement

Azzeddine
RIGAT

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

Syntax:

```
for (initialization; condition; incrementation) statement
```

Example:

```
for (var i = 1, j = 1; i <= 4; i++,
j+=2) {
   console.log('i = ' + i + ' j = ' +
j);
}
```

Loop statements, The for-in statement

Azzeddine
RIGAT

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

Syntax:

```
for (variable in expression) statement
```

Example:

```
var michel = {                  // michel is an object
   familyName:'Buffa',          // familyName, givenName, age
                                // are its properties
   givenName: 'Michel',
   age: 51
}
for(var property in michel) {   // the for-in will
                                // enumerate properties
   console.log(property);       // will print "familyName",
                                // "givenName",
                                // "age"
   console.log(michel[property]);  // michel['givenName'] same
                                   // as michel.givenName
}
```

Loop statements, The continue statement

Azzeddine
RIGAT

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:

```
continue
```

Example:

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

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...*

Loop statements, The break statement

Azzeddine
RIGAT

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

Syntax:

```
break
```

Example:

```
var tab = ['michel', 'john', 'donald', 'paul']; // johh at index = 1

function isNameInTheArray(name, theArray) {
  console.log("Number of elements in the array : " + theArray.length);

  for(var i=0; i < theArray.length; i++) {
    console.log('comparing with element in the array at pos ' + i);

    if(theArray[i] === name) {
      console.log('the name ' + name +
                  ' is in the array at pos: ' + i);
      break;
    } else {
      console.log(name + ' is not at pos ' + i);
    }
  }
}

// Execute the function
isNameInTheArray('john', tab);
```

Loop statements, The break statement

Azzeddine
RIGAT

Hint: We can have the same result without coding that much of lines if we know arrays methods

Voila: ;)

```
var tab = ['michel', 'john', 'donald', 'paul'];
// johh at index = 1

tab.indexOf('john')
```

Caution: Syntax from one programming language to another different, for example:

JavaScript:

```
tab.indexOf('john')
tab.includes('john')
```

Python :

```
tab.index('john')
'john' in tab
```

Loop statements, The break statement

Azzeddine
RIGAT

Copy and paste the break example on **page 25** 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).

Loop statements, Projects

Azzeddine
RIGAT

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 **example1.7** in the first module, the math function plotter example used input fields. Or look at the section about DOM in this module.

Loop statements, Projects

Azzeddine
RIGAT

Optional projects:

- 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

Functions and callbacks….PartII

Azzeddine
RIGAT

Two ways to declare a function

1 - Standard function declaration

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

```
function functionName(parameters) {
   // code to be executed
   }
```

A function declared this way can be **called** like this:

```
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. Check **example.II.2.3.0**

Functions and callbacks….PartII

Azzeddine RIGAT

HTML

```
<!doctype html>
<html>
      <head lang="en">
            <title>Function part I</title>
            <meta charset="utf-8">
      </head>
      <body>
      </body>
</html>
```

JS

```
function sum(a, b) {
   // this function returns a result
   return (a + b);
}
 function displayInPage(message, value) {
  // this function does not return anything
  document.body.innerHTML += message + value + "<br>";
}
 var result = sum(3, 4);
displayInPage("Result: ", result);
 // we could have written this
displayInPage("Result: ", sum(10, 15));
```

Functions and callbacks….PartII

Azzeddine RIGAT

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. Check **example.II.2.3.1**

HTML

```
<!doctype html>
<html>
      <head lang="en">
            <title>Function part 1</title>
            <meta charset="utf-8">
      </head>
      <body>
      </body>
</html>
```

JS

```
var sum = function(a, b) {
   return (a + b);
};
 var displayInPage = function(message, value) {
  // this function does not return anything
  document.body.innerHTML += message + value + "<br>";
};
 var result = sum(3, 4);
displayInPage("Result: ", result);
 // we could have written this
displayInPage("Result: ", sum(10, 15));
```

Functions and callbacks….PartII

Azzeddine
RIGAT

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"**.

Functions and callbacks

Azzeddine
RIGAT

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). Check **example.II.2.3.2**

Functions and callbacks

Azzeddine
RIGAT

HTML

```
<!doctype html>
<html>
      <head lang="en">
            <title>Function part I</title>
            <meta charset="utf-8">
      </head>
      <body>
            <p>Click in the page!</p>
      </body>
</html>
```

JS

```
window.addEventListener('click', processClick);

function processClick(event) {
 document.body.innerHTML += "Button clicked<br>";
}

// We could have written this, with the body of the
callback as an argument of the addEventListener function
window.addEventListener('click', function(evt) {
 document.body.innerHTML += "Button clicked version
2<br>";
});
```

Handling events

Azzeddine
RIGAT

Adding interactivity to a Web application can only be achieved with CSS, using the :hover pseudo CSS class, for instance. For example: check **example.II.2.4.0**

HTML

```
<!DOCTYPE html>
<html lang="en">
      <head>
            <title>Mouse over</title>
            <meta charset="utf-8"/>
            <link href='example.II.2.4.0.css'
           rel=stylesheet>
      </head>
      <body>
            <button>Put the mouse cursor over me</button>
      </body>
</html>
```

CSS

```
button:hover {
 color:red;
 border:2px solid lightgreen;
}
```

Handling events

Azzeddine
RIGAT

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 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".

Handling events, Adding and removing event listeners

Azzeddine RIGAT

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): check **example.II.2.4.1**

HTML

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <title>First example of an event listener</title>
 <meta charset="utf-8">
 <script>
  addEventListener('click', function(evt) {
        document.body.innerHTML += "Button clicked!<br>";
   });
</script>
</head>
<body>
 <p>Click anywhere on this page</p>
</body>
</html>
```

Handling events, Adding and removing event listeners

Azzeddine RIGAT

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). check **example.ll.2.4.2**

HTML

```
<!DOCTYPE html>
<html lang="en">
   ...
<body>
<button id="myButton">Click me!</button>
<p></p>
<script>
  var b = document.querySelector("#myButton");
  b.addEventListener('click', function(evt) {
    alert("Button clicked");
  });
</script>
</body>
</html>
```

Handling events, Adding and removing event listeners

Azzeddine RIGAT

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

- 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");*
- 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** = "....")

Handling events, Adding and removing event listeners

Azzeddine RIGAT

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

HTML

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <title>Third example of an event listener</title>
   <meta charset="utf-8">
     <script>
        function processClick(evt) {
         alert("Button clicked");
        };
 </script>
</head>
<body>
<button id="myButton" onclick="processClick(event);">Click
me!</button>
</body>
</html>
```

Handling events, Adding and removing event listeners

Azzeddine RIGAT

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. check **example.II.2.4.4**

HTML

```
<html lang="en">
      ...
<body>
 <button id="myButton">Click me, this will work only
once!</button>
 <p></p>
 <script>
   var b = document.querySelector("#myButton");
   b.addEventListener('click', processClick);
   function processClick(evt) {
    alert("Button clicked, ….");
    b.removeEventListener('click', processClick); }
 </script>
</body>
</html>
```

Handling events, The event object

Azzeddine
RIGAT

The event object is the only parameter passed to event listeners

Typical example:

```
function processClick(evt) { alert("Button clicked!");}
```

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.

Handling events, The event object

Azzeddine
RIGAT

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.

Handling events, The event object

Azzeddine
RIGAT

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.

Handling events, Page lifecycle events

Azzeddine
RIGAT

Page lifecycle events; These events detect when the page is loaded and when the DOM is ready. 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). Using **document.onload = function;** inside the Js file or '**onload = function();**' inside the body tag.
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 - window.innerHeight;** var percent = **(window.pageYOffset / max);** ...to know the percentage of the scroll in the page.

Handling events, Page lifecycle events

Azzeddine
RIGAT

- Example 1: wait until the page is loaded (when the DOM is ready) before doing something
  - **example.II.2.4.6;** this first variant that uses **<body onload="init();">**
  - **example.II.2.4.7;** this second variant: using **window.onload = init;** in the JavaScript code...
- Example 2: detect a resize of the window
  - **example.II.2.4.8;** 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: **example.II.2.4.9;** do something as the page is being scrolled up or down
  - **document.body.scrollHeight** is the **body height**
  - **window.innerHeight** is the current **window height**
  - **evt.pageY** is the **current position of the scroll** and equivalent to **window.pageYOffset**

Handling events, Key events

Azzeddine
RIGAT

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:

```
window.addEventListener('keydown', function(event) {
    if (event.keyCode === 37) {
        //the value "37" is the key code that corresponds to the left arrow
        //left arrow was pressed
    }
});
```

Handling events, Key events

Azzeddine
RIGAT

The different **key events**:

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).

Handling events, Key events

Azzeddine
RIGAT

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.

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

Handling events, Key events

Azzeddine RIGAT

Example 1: use keyup and keydown on the window object; Check **example.II.2.4.10.keyEvents**.

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="utf-8">
      ...
</head>
 <body>
   <p>Please type some keys and see what
happens</p>
   <div id="keys"></div>
 </body>
</html>
```

JS

```
window.onkeyup = processKeyUp;
window.onkeydown = processKeyDown;

function processKeyUp(evt) {
 var keys = document.querySelector('#keys');
 keys.innerHTML += "keyup, code: " + evt.keyCode +
"<br>";
}

function processKeyDown(evt) {
 var keys = document.querySelector('#keys');
 keys.innerHTML += "keydown, code: " + evt.keyCode +
"<br>";
}
```

Handling events, Key events

Azzeddine
RIGAT

Assignment; Improve **example.II.2.4.10.keyEvents**.

- Use event.type, and reduce the functions declaration from two to just one function
- What is the difference between keyCode of c and C ?, and how can we differentiate between them ?
- Add a feature that your script will clean the window every time you press on c.
- Could you display the event name on your browser for every event you make from mouse or keyboard ? if not, explain the reason.

Handling events, Key events

Azzeddine RIGAT

Example 2: detect a combination of keys + modifier keys (shift, ctrl, alt); Check **example.II.2.4.11.keyEvents**. Try to type shift-a for example, ctrl-shift-b or alt-f...

HTML

```
<!DOCTYPE html>
<html lang="en">
      ...
</head>
 <body>
   <p>Please type some keys and see what
happens. Try typing key modifiers at the
same time: shift, alt, control</p>
   <div id="keys"></div>
 </body>
</html>
```

JS

```
window.onkeydown = processKeyDown;
function processKeyDown(evt) {
 var keys = document.querySelector('#keys');
   keys.innerHTML += "keypress, code: " + evt.keyCode + "
Modifiers : ";

 var modifiers = "";
  if(event.shiftKey)
   modifiers += "SHIFT ";
      ...
  keys.innerHTML += modifiers + "<br>";
}
```

Handling events, Key events

Azzeddine
RIGAT

Assignment; Improve **example.II.2.4.11.keyEvents**.

- Does't necessary to check all the modifier every time ??
- Could we use switch to improve the performance of our script ? if not, what is the reason ?
- Provide an alternative way to improve the performance, that we don't necessary check all the modifier every time.

Handling events, Dealing with different keyboard layouts

Azzeddine RIGAT

Please do not assume that each key is at the same location on the keyboard for every language! 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 language to another. Extract from the "*Internationalise 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.

There was the wonderful WebGL 2 technological demo After The Flood, and the very cute Alpaca Peck. Shaw 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.**"

Handling events, Dealing with different keyboard layouts

Azzeddine
RIGAT

QWERTY layout, used in US, GB, etc:

AZERTY layout, used in French-speaking countries:

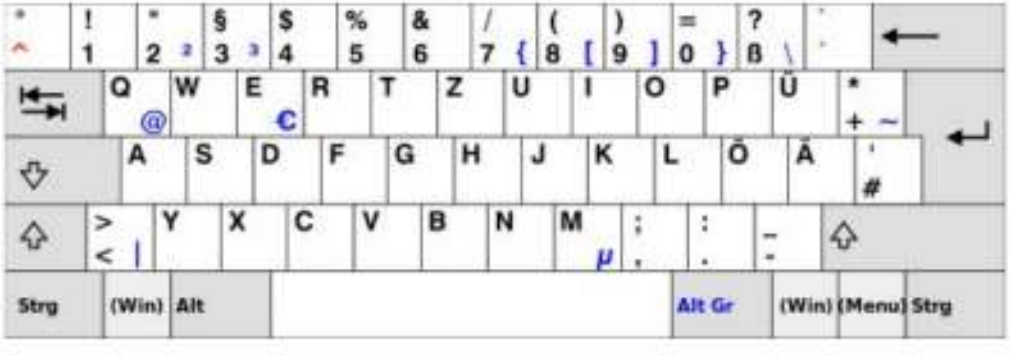
Handling events, Dealing with different keyboard layouts

Azzeddine
RIGAT

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

DVORAK:

QWERTZ:



Handling events, Dealing with different keyboard layouts

Azzeddine
RIGAT

Arabic keyboard:

Saudi Arabian (Arabic 101) Keyboard Layout

Bangla National (Jatiyo) keyboard:

Handling events, Key and code properties

Azzeddine
RIGAT

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

You may have noticed that in some examples from the previous pages about key events, we used event.keyCode in order to display the character code that has been typed. All major browsers have implemented three very practical ***key properties***.

- **event.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'.**Usage:** differentiate between upper and lower characters.
- **event.keyCode:** 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. **Usage:** get a number for every key.
- **event.code**: gives you 'KeyQ' while event.key gives you 'q'.**Usage:** This one is not helpful, just in one case if you wanna now if you are pressing on right of left modifier ex;alterLeft and alterRight.

Handling events, Key and code properties

Azzeddine
RIGAT

Caution:

- 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.

- **Be careful if you are going to use event.code**

Handling events, Key and code properties

Azzeddine
RIGAT

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. You can read about the existing mechanical layouts around the world, as well as their reference keyboard. For instance here is their reference keyboard for the alphanumerical part:

Handling events, Key and code properties

Azzeddine
RIGAT

Also read through the examples given in the specification. They show very clearly what happens when the user presses various types of keys, both for code and key. Check
example.ll.2.4.12.key.code.properties

HTML

```
<html lang="en">
    ...
 <body>
  <p>Press some keys on your keyboard and
see the corresponding evt.key and evt.code
values. If you are not using a QWERTY
keyboard, notice that the values might be
different. This is because an 'a' on an
AZERTY keyboard, will correspond to the
KeyQ code on the reference keyboard.</p>
   <p>  You typed:</p>
 </body>
</html>
```

JS

```
window.onkeydown = function(evt) {
 document.body.innerHTML += "key = " + evt.key + "<br>";
document.body.innerHTML += "code = " + evt.code +
"<br><br>";
}
```

Handling events, Key and code properties

Azzeddine
RIGAT

Tasks: Change the language, precisely choose one of the languages we have mentioned and see the difference.

Handling events, Mouse events

Azzeddine
RIGAT

Mouse interaction, mouse events:

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*

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):

```
canvas.addEventListener('mousedown', function (evt) {
  // do something with the mousedown event
});
canvas.addEventListener('mouseup', function (evt) {
  // do something with the mouseup event
});
```

Handling events, Mouse events, Event types related to mouse

Azzeddine
RIGAT

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

Handling events, Mouse events, Event types related to mouse

Azzeddine RIGAT

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 shift" key was pressed when an event was triggered
details	Returns a number that indicates how many times the mouse was clicked

Handling events, Mouse events, Event types related to mouse

Azzeddine RIGAT

Check example **example.II.2.4.13.mouseEvents;** that detects a click on an element

HTML

```
      ...
</head>
 <body>
   <button id='button'
onclick="processClick(event)">Button</butt
on>
   <div id='myDiv'
onclick="processClick(event)">Click also
on this div!</div>
   <div id='clicks'></div>
 </body>
</html>
```

JS

```
window.onclick = processClick;
function processClick(event){
   var clicks = document.querySelector("#clicks");
   var target = event.target.id ;
   if (target === ""){
       clicks.innerHTML += "... not on a particular ..<br>"
   }
   else{
   clicks.innerHTML += "Element clicked id: " + target +
"<br>";
   }
   event.stopPropagation();//Experiment what will happened
if you comment this line
}
```

Handling events, Mouse events, Event types related to mouse

Azzeddine
RIGAT

Assignments: Detect a click on an element

- Implement the same example as in example.II.2.4.13.mouseEvents with the same result, but without the event.stopPropagation() function
- Write down a script that show the differences between ScreenX/Screen, clientX/clientY and pageX/pageY

Handling events, Mouse events, Event types related to mouse

Azzeddine RIGAT

Check example **example.II.2.4.14.mouseEvents**, that detect a mousemove and get the mouse position relative to the element that fired the event

HTML

```
    ...
 <body>
      <h1>Please move the mouse on the
grey canvas below!
      </h1>
      <p>You'll see that the displayed
position is relative the canvas:
          (0, 0) is now at the top left
corner of the canvas,
          even if we scroll the page!
      </p>
      <canvas id="myCanvas" width=300
height=50></canvas>
      <div id="mousePositions"></div>
```

JS

```
window.onload = init;

function init(event){
   canvas = document.querySelector("#myCanvas");
   canvas.onmousemove = processMouseMouveCanvas;
}
function processMouseMouveCanvas(event){
   var mousePosition =
document.querySelector("#mousePositions");
   mousePosition.innerHTML = "mouse pos X: " +
event.clientX + " mouse pos Y: " + event.clientY
}
```

Handling events, Mouse events, Event types related to mouse

Azzeddine
RIGAT

Observation:

As you can see in example.II.2.4.14.mouseEvents that when you place the mouse cursor on the top left of the canvas you get coordinate different form (0, 0). To fix this issue we need to get the coordinate of our canvas using the function **getBoundingClientRect()**. The latter function returns a rect(left, top, width, height). example.II.2.4.15.mouseEvents, solve that is burden.

Handling events, Mouse events, Event types related to mouse

Azzeddine
RIGAT

Check example **example.II.2.4.16.mouseEvents,** that combine mouseup, mousedown, mousemove to implement a click and drag behavior, after you run it. the way its works is freaky when you just click down on you the grey rectangle to move it at all position, except at the top left position which works normal.

Reason: when we call `selected.style.left = currentPosition` we do not move our rectangle, instead we are jumping it to the current position of the mouse pointer. Our goal it to move smoothly our rectangle, which mean that you need to first calculate the difference between the old position and the distance we move our mouse, secondly add that difference and add it to your rectangle rect object, finally update the displayed position to the rect position. I fixed that behavior in **example.II.2.4.17.mouseEvents.**

Note: do not forget to set `position: absolute;` in your div inside the CSS file, otherwise your div can't move.

Handling events, Mouse events, Event types related to mouse

Azzeddine
RIGAT

Check example **example.II.2.4.18.mouseEvents,** that creates and attaches a right-click context menu to any element

Assignment: Reduce the length of the code and let it clean by removing unnecessary lines and replace lines of code with shorter length.

Handling events, Form and input field events, Events related to forms

Azzeddine
RIGAT

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**

Handling events, Form and input field events

Azzeddine RIGAT

Check example **example.II.2.4.19.formEvents;** validating on the fly as the user types in a text input field

HTML

```
<p>Just type a name in the input field and see what happens!</p>
<label>
 <span>Name (required) :</span>
 <input type="text"
        name="nom"
        maxlength="32"
        required
        oninput = "validateName(this)"
        id='name'>
</label>
 <p>
 <span id="nameTyped"></span>
</p>
```

JS

```
function validateName(field)
 // this is the input field text content
 var name = field.value;
  // get the output div
 var output = document.querySelector('#nameTyped');
 // display the value typed in the div
 output.innerHTML = "Valid name: " + name;
  // You can do validation here, set the input field to
 // invalid is the name contains forbidden characters
 // or is too short
 // for example, let's forbid names with length < 5 chars
 if(name.length < 5) {
   output.innerHTML = "This name is too short (at least 5 chars)"; } }
```

Handling events, Form and input field events

Azzeddine
RIGAT

Assignment:

Redo the JS file of **example.II.2.4.19.formEvents;** using the old way without the help of ***this***, introduced in the oninput.

Handling events, Form and input field events

Azzeddine RIGAT

Check example **example.II.2.4.20.formEvents;** do something while a slider is being moved

HTML

```
<body>
 <h1>Simple <code>&lt;input
type=range&gt;</code> field validation
using the 'input' event</h1>
 <p>Just move the slider</p>
<label>
 <input type="range"
        min=1
        max=12
         step=0.1
        oninput = "doSomething(event)">
</label>
 <p>
 <span id="sliderValue"></span> </p>
```

JS

```
function doSomething(evt) {
        // this is the slider value
        var val = evt.target.value;
         // get the output div
        var output = document.querySelector('#sliderValue');
        // display the value typed in the div
        output.innerHTML = "Value selected: " + val;
}
```

Handling events, Form and input field events

Azzeddine
RIGAT

Assignment:

Modify the JS file in **example.II.2.4.20.formEvents;** to change the background of 'Just move the slider' between 0 and 255 value of the green color in the rgb palette. Like this one:

Simple `<input type=range>` field validation using the 'input' event

Just move the slider

0 255

Value selected: 211

Handling events, Form and input field events

Azzeddine RIGAT

Check example **example.II.2.4.21.formEvents;** detect the change in a number input field

HTML

```
<p>type a number or use the small vertical
arrows</p>
<label>
 Type a number:
<input type="number"
        min=1
        max=12
         step=0.1
       oninput = "doSomething(event)">
</label>
 <p>
 <span id="numberValue"></span>
```

JS

```
function doSomething(evt) {
        // this is the slider value
        var val = evt.target.value;
         // get the output div
        var output = document.querySelector('#numberValue');
        // display the value typed in the div
        output.innerHTML = "Value selected: " + val;
}
```

Handling events, Form and input field events

Azzeddine RIGAT

Check example **example.II.2.4.22.formEvents;** choose a color and do something

HTML

```
 <p>Pick a color to change the background
color of the page</p>
<label>
<input type="color"
       onchange =
"changePageBackgroundColor(this.value);">
 <!-- we could have used oninput= in the
previous line -->
</label>
 <p>
 <span id="choosedColor"></span>
</p>
```

JS

```
function changePageBackgroundColor(color) {
    document.body.style.backgroundColor = color;
     // get the output div
     var output =
    document.querySelector('#choosedColor');
     // display the value typed in the div
     output.innerHTML = "Color selected: " + color;
}
```

Handling events, Recap

Azzeddine
RIGAT

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

```
<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.

Handling events, Recap

Azzeddine
RIGAT

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

```
document.getElementById('someDiv').onclick = function() {
 alert('clicked!');
}
```

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)

```
document.getElementById('someDiv').addEventListener('click', function() {
 alert('clicked!');
}, 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).

Handling events, Recap

Azzeddine
RIGAT

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:

```
element.addEventListener('click', function(event) {
 // now you can use event object inside the callback
}, 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, I will remind you how to deal with the keyboard and the mouse.

Handling events, Projects

Azzeddine
RIGAT

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. 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 this course topic, you will see how to work with graphics and animation, you can also imagine a graphic version of this game.

The DOM API, Introduction

Azzeddine
RIGAT

When a user clicks on a link or enters a URL in the address of the 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**.

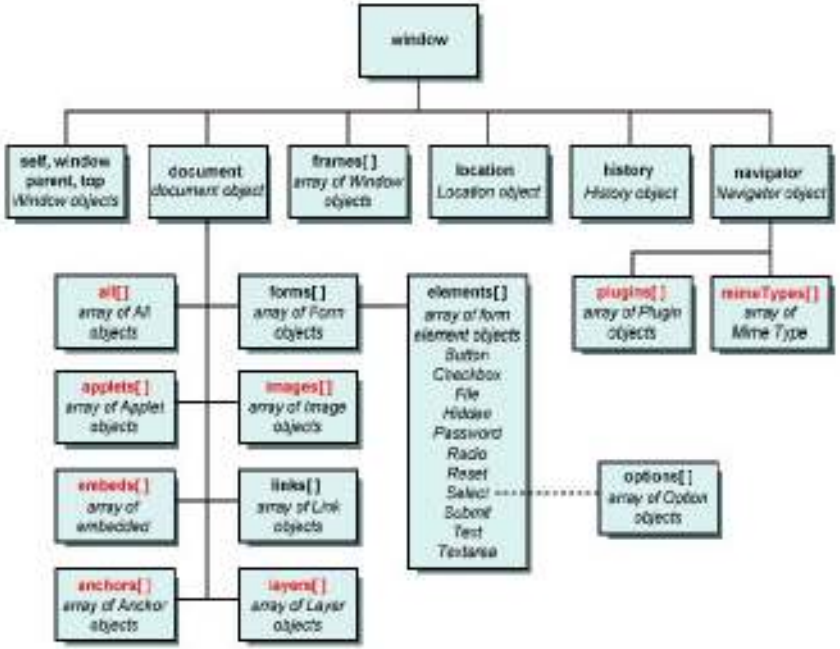
The DOM API is a programming interface the JavaScript programmer can use to modify the HTML content or the CSS style of HTML elements on the fly.

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

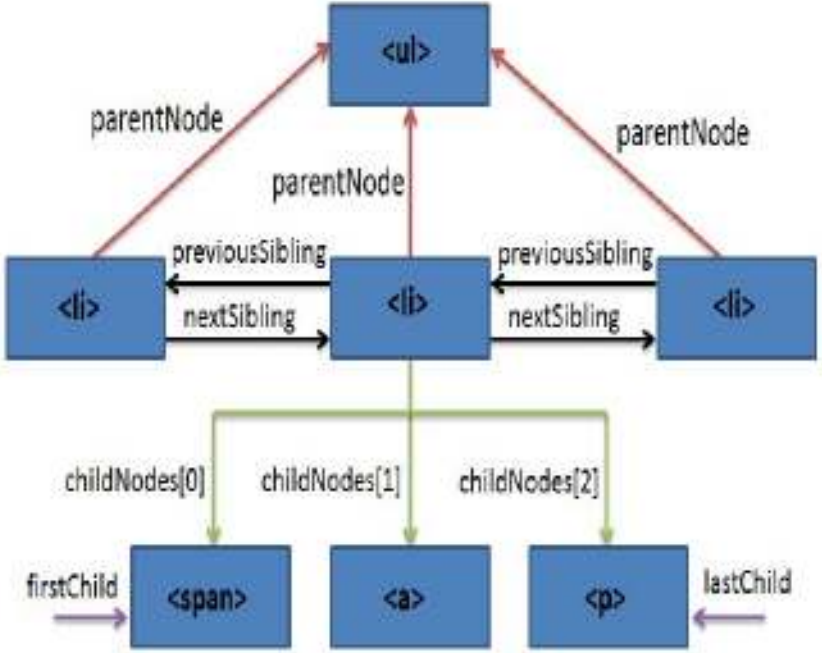
The DOM API, Introduction

Azzeddine RIGAT

window's DOM



unordered list DOM



The DOM API, Introduction

Azzeddine
RIGAT

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.

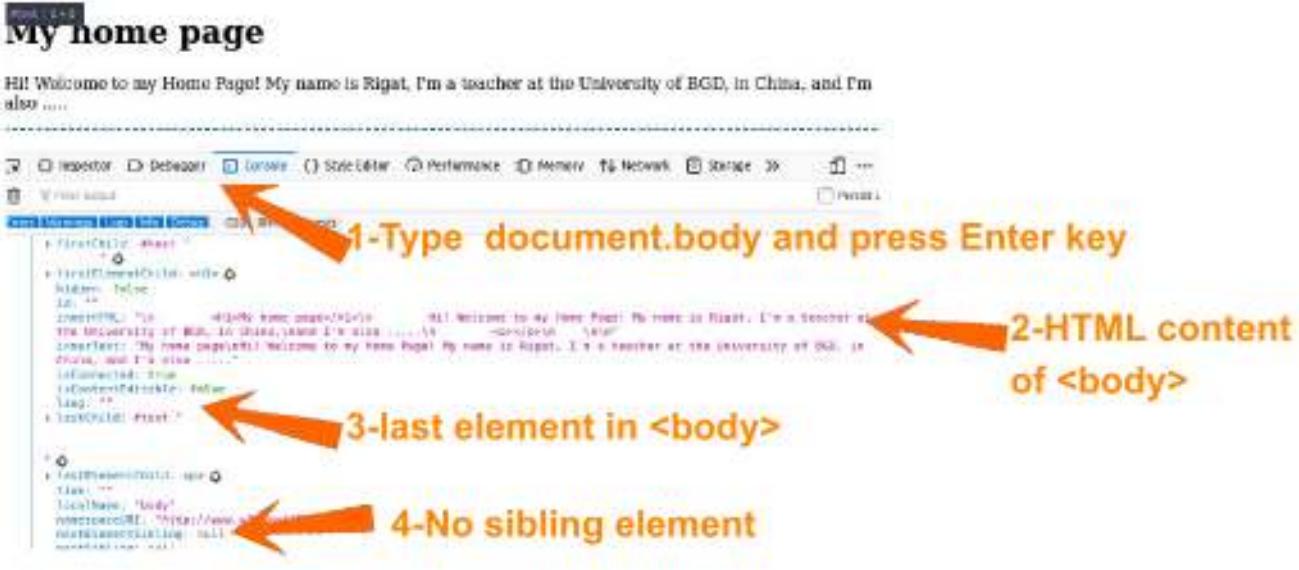
In JavaScript, the DOM is accessible through the property **document** of the global object window. We rarely manipulate the window object directly as it is implicit: **window.document** is the same as **document**.

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

The DOM API, Introduction

Azzeddine
RIGAT

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:



The DOM API, Introduction

Azzeddine
RIGAT

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: Check example **example.II.2.5.00.DOM**.

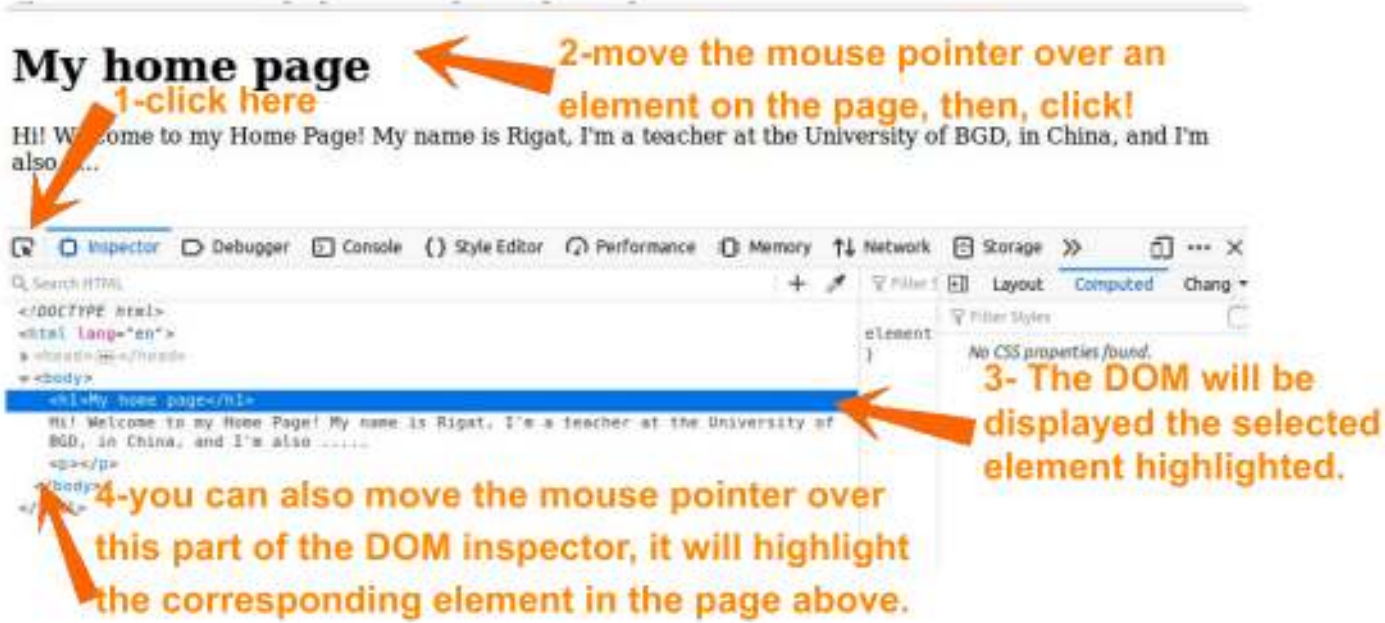
My home page

Hi! Welcome to my Home Page! My name is Rigat, I'm a teacher at the University of BGD, in China, and I'm also ....

The DOM API, Introduction

Azzeddine
RIGAT

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:



The DOM API, Introduction

Azzeddine
RIGAT

A warning about the DOM API:

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.

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

There are two ways, the old fashion way and selector API, here I am going to start with the modern one, because it has just mainly two methods that could replace all the old ones.

1 - With the selector API (recommended)

Extract from HTML5 **selectors API** -- It's like a Swiss Army Knife for the DOM *: "One of the many reasons for the success of JavaScript libraries like **jQuery** and **Prototype**, 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

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

Ah... these methods owe a lot to jQuery! 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.

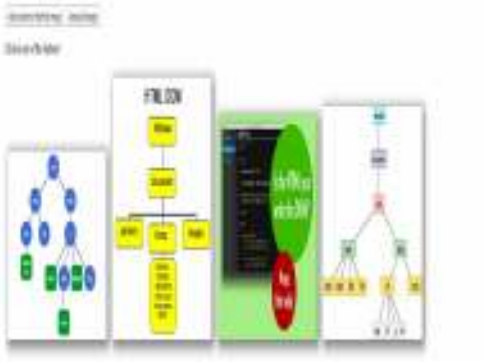
The DOM API, Accessing HTML elements

Azzeddine
RIGAT

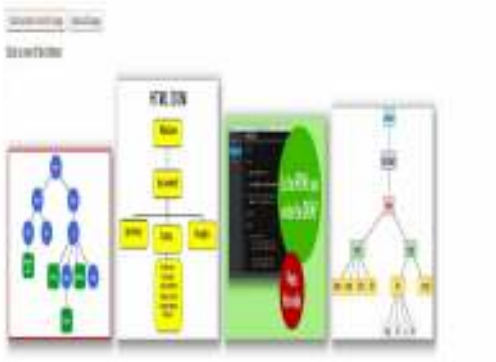
Typical use:

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:Check **example.II.2.5.01.DOM**.

Onload



After we click on the first button



After we click on the second button

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

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="images/dom1.png"
12          id="img1"
13          width=400>
14  <img src="images/dom2.gif"
15          width=400>
16  <img src="images/dom3.jpg"
17          width=400>
18  <img src="images/dom4.png"
19          width=400>
20  ...
```

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

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 3-8). The two other functions are called when one of the HTML buttons is clicked (line 10 and line 15).

```
1  window.onload = init;
2
3  function init(event){
4      images = document.querySelectorAll('img');
5      images.forEach( image=> {
6          image.style.boxShadow = "5px 5px 15px 5px grey";
7          image.style.margin = "10px"});
8  }
9
10 function addBorderToFirstImage(event){
11     var firstImage = document.querySelector('#img1');
12     firstImage.style.border = "2px solid red";
13 }
14
15 function resizeAllImages(event){
16     images.forEach(image =>{
17         image.style.width = "100px";
18     });
19 }
```

Miscellaneous examples of use of **querySelector(CSSSelector)** and **querySelectorAll(CSSselector)**

Where are going to see some other examples that use more complicated CSS selectors

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

Check **example.II.2.5.02.DOM,** get all <li> directly in a <ul> of class nav:

HTML

```
...
<body>
    <button
onclick='firstLiClassRedInUl()'>Select
first li of class red and color it in
red</button>
    <button
onclick='allLiInUlClassNav()'>Underline
All li in a ul of class nav</button>
    <ul class='nav'>
        <li>Home</li>
        <li class='red'>Products</li>
        <li>About</li>
    </ul>
...
```

JS

```
function firstLiClassRedInUl(event){
var first_li = document.querySelector(".red");
first_li.style.color = 'red';
}

function allLiInUlClassNav(event){
    var all_li = document.querySelectorAll("ul.nav li")
    all_li.forEach(li => {
        li.style.textDecoration = 'underline';
    })
}
```

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

Assignment:

Write down a script that displays all checked <input type="checkbox"> elements located inside an element of a given id, then uncheck the checked elements and remove the text when click on the reset button

Onload

After we click on the first button

After we click on the second button

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

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
4  // all li in a ul, but only even elements
5  var els = document.querySelectorAll('ul li:nth-child(even)');
6
7  // all td directly in tr in a form of class test
8  var els = document.querySelectorAll('form.test > tr > td');
9
10 // all paragraphs of class warning or error
11 querySelectorAll("p.warning, p.error");
12
13 // first element of id=foo or id=bar
14 querySelector("#foo, #bar");
15
16 // first p in a div id=bar
17 var div = document.getElementById("bar");
18 var p = div.querySelector("p");
19 //equivalent to the last one
20 var p = document.querySelector("#bar > p")
21 //equivalent as well to
22 var p = document.querySelector("#bar p")
```

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

2 - The second option 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');

The DOM API, Accessing HTML elements

Azzeddine
RIGAT

- **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.

The DOM API, Changing the style of selected HTML elements

Azzeddine RIGAT

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

- **Remove the "-" sign,**
- **Capitalize the word after the "-" sign!**

Simple, isn't it?

Examples:

text-align becomes *style.textAlign*

margin-left becomes *style.marginLeft*

etc.

The DOM API, Changing the style of selected HTML elements

Azzeddine RIGAT

We've already seen many examples in which we selected one or more elements, and modified their content. Let's summarize 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

1. **Using the innerHTML property**

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

2. **Using the textContent property**

Open example **example.II.2.5.04.DOM**, then open the devtool console.

Here, we extract from the **HTML code**:

```
<p id="first">first paragraph</p>
<p id="second"><em>second</em> paragraph</p>
```

The DOM API, Changing the style of selected HTML elements

Azzeddine RIGAT

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 *...* that surrounds it. However, when we modify the textContent value, it also replaces the text decoration (the is removed), this is done at *lines 16-20.*

```
1  window.onload = init;
2
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"
8
9      firstP.textContent = "Hello I'm the first paragraph";
10     console.log(firstP.textContent); // "Hello I'm the first paragraph"
11
12     var secondP = document.querySelector("#second");
13     console.log(secondP.textContent); // "second paragraph"
14     console.log(secondP.innerHTML);   // "<em>second</em> paragraph"
15
16     secondP.textContent = "Hello I'm the second paragraph";
17     console.log(secondP.textContent); // "Hello I'm the second
18                                       // paragraph"
19     console.log(secondP.innerHTML);   // "Hello I'm the second
20                                       // paragraph"
21 }
```

The DOM API, Adding new elements to the DOM

Azzeddine
RIGAT

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:

```
var elm = document.createElement(name_of_the_element);
```

Examples:

```
var li = document.createElement('li');
var img = document.createElement('img'); etc.
```

2. **Set some attributes / values / styles for this element.**

Examples:

```
li.innerHTML = '<b>This is a new list item in bold!</b>'; // can add HTML in it
li.textContent = 'Another new list item';
li.style.color = 'green'; // green text
img.src = "http://..../myImage.jpg"; // url of the image
img.width = 200;
```

3. Add the newly created element to another element in the DOM, using **append()**, **appendChild()**, **insertBefore()** or the **innerHTML** property

The DOM API, Adding new elements to the DOM

Azzeddine
RIGAT

Examples:

```
var ul = document.querySelector('#myList');
ul.append(li); // insert at the end, appendChild() could also be used (old)
ul.prepend(li); // insert at the beginning
ul.insertBefore(li, another_element_child_of_ul);// insert in the middle
document.body.append(img); // adds the image at the end of the document
```

As an example running on your browser check **example.II.2.5.05.DOM**.

The DOM API, Adding new elements to the DOM

Azzeddine RIGAT

HTML

```
<body>
   <label for="newNumber">Please enter a number</label>
       <input type="number" id="newNumber" value=0>
       <button onclick="add();">Add to the list</button>
       <br>
       <button onclick="reset()">Reset list</button>

   <p>You entered:</p>
   <ul id="numbers"></ul>
</body>
```

JS

```
function add() {
 var val =
document.querySelector('#newNumber').value;
 if((val !== undefined) && (val !== "")) {
   var ul = document.querySelector("#numbers");
   // add it to the list as a new <li>
   var newNumber = document.createElement("li");
   newNumber.textContent = val;
   // or newNumber.innerHTML = val
   ul.append(newNumber);
 }
}
function reset() {
 var ul = document.querySelector("#numbers");
 // reset it: no children
 ul.innerHTML = ""; }
```

The DOM API, Adding new elements to the DOM

Azzeddine
RIGAT

Here is an abbreviated form, using the innerHTML property;

```
ul.innerHTML += "<li>" + val + "</li>";
```

Replace the above line with with lines 8-12, inside the JS file of **example.II.2.5.05.DOM.**

The DOM API, Moving HTML elements in the DOM

Azzeddine
RIGAT

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

```
var li = createElement('li');
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: **example.II.2.5.06.DOM**

The DOM API, Moving HTML elements in the DOM

Azzeddine RIGAT

HTML

```
<body>
    <p>
        Click on a browser icon to move it the box of cool browsers:
    </p>
    <div>
        <img alt='chrome browser' id='chrome' src='images/chrome.png' onclick="move(this);">
        <img alt='firefox browser' id='firefox' src='images/firefox.png' onclick="move(this);">
        ...
    </div>
    <div class='box'>
        <p> Cool Web browsers </p>
    </div>
</body>
```

JS

```
function move(image){
    // find the place where I will put it
    var box = document.querySelector('.box');
    // move that image to box class
    box.append(image);
    // desactive the click event
    image.onclick = null;
}
```

The DOM API, Moving HTML elements in the DOM

Azzeddine
RIGAT

Another, more significant example, that also uses drag'n'drop, check **example.II.2.5.07.DOM**

HTML

```
....
          <img alt='safari browser' id='safari'
src='images/safari.png' ondragstart="drag(this,
event);">
        </div>
        <div id='coolBrowsers' ondrop="drop(this,
event);" ondragover="return false">
            <p> Cool Web browsers </p>
        </div>

        <div id='oldBrowsers' ondrop="drop(this, event);"
ondragover="return false">
            <p> Cool Web browsers </p>
        </div>
```

JS

```
function drag(image, event){
 event.dataTransfer.setData("browser",
image.id);
}

function drop(target, event){
 var id = event.dataTransfer.getData('browser');

target.append(document.querySelector("#" + id));
event.preventDefault();
}
```

The DOM API, Moving HTML elements in the DOM

Azzeddine
RIGAT

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.

The DOM API, Removing elements from the DOM

Azzeddine
RIGAT

Removing elements using the **removeChild() or remove()** methods. Let's take an example that we've already encountered. This time, you will check the elements you want to remove from the list. Check **example.II.2.5.08.DOM**

HTML

```
<body>
    <button onclick="removeCheckedItems();">Remove Checked items</button>
    <section id='fruits'>
        <label id='apples'> <input type='checkbox' value='apples'>Apples </label>
        <label id='oranges'><input type="checkbox" value='oranges'>Oranges</label>
        <label id='bananas'><input type='checkbox' value='bananas'>Bananas</label>
        <label id='grapes'><input type='checkbox' value='grapes'>Grapes</label>
    </section>
</body>
```

The DOM API, Removing elements from the DOM

Azzeddine
RIGAT

JS v1

```
function removeCheckedItems(event){
   var listOfSelectValues = '';
   var list = document.querySelectorAll('#fruits
input:checked');
   list.forEach(checked => {
       //find it label first
       var label =
document.querySelector("#"+checked.value);

       //remove the label elment
       label.remove();
   });
}
```

JS v2

```
function removeCheckedItems(event){
   var listOfSelectValues = '';
   var list = document.querySelectorAll('#fruits
input:checked');
   var fruits = document.querySelector('#fruits');
   list.forEach(checked => {
      var parentNode = checked.parentNode;
      console.log(parentNode);

      fruits.removeChild(parentNode);

   });

}
```

Introduction to drawing/animating,

Azzeddine RIGAT

Introduction to drawing

When you talk about **animating draws/images** or **building games** we need to use a standard HTML5 API that is Canvas. Here I am going to introduce HTML5 **Canvas** basic usage

The **HTML5 canvas** is a transparent element that is useful for drawing and animating. We'll see some simple examples here, as we're going to finish this part of the course by writing a small, simple game together, that will use most of what we've learnt so far: **loops**, **conditional statements**, **events**, **functions**, **callbacks**, **simple objects**, a few **input fields**, etc.

A typical HTML code for adding a canvas to a Web page:

```
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <title>Make my game in a canvas</title>
6      </head>
7      <body>
8          <canvas id="myCanvas" width="200" height="200"></canvas>
9      </body>
10 </html>
```

The canvas declaration is at line 8. Use attributes to give it a `width` and a `height`, but unless you add some CSS properties, you will not see it on the screen because it's transparent!

Introduction to drawing/animating,

Azzeddine RIGAT

Introduction to drawing

Let's use CSS to reveal the canvas, for example, add a 1px black border around it:

```
canvas {
    border: 1px solid black;
    }
```

And here is a reminder of best practice when using the canvas:

- Use a function that is called AFTER the page is fully loaded (and the DOM is ready), select the canvas in the DOM (this is the init function (as in window.onload = init) we already saw many times).
- Then, get a 2D graphic context for this canvas (the context is an object we will use to draw on the canvas, to set global properties such as color, gradients, patterns and line width).
- Only then can you can draw something,
- Do not forget to use global variables for the canvas and context objects. I also recommend keeping the width and height of the canvas somewhere. These might be useful later.
- For each function that will change the context (color, line width, coordinate system, etc.), start by saving the context, and end by restoring it.

Introduction to drawing/animating,

Azzeddine RIGAT

Introduction to drawing

First example: check **example.II.2.6.00.Canvas**

JS

```
var canvas, ctx;
window.onload = function init(){
   canvas = document.querySelector('#myCanvas');
     // This is the object that allow us to draw
   ctx = canvas.getContext('2d');
   ctx.fillStyle = 'red';      // filled rectangle
   ctx.fillRect(10, 10, 30, 30);
   ctx.strokeStyle = 'green'; // wireframe rectangle
   ctx.lineWidth = 4;
   ctx.strokeRect(100, 40, 40, 40);
   // fill circle, will use current ctx.fillstyle
   ctx.beginPath();
   ctx.arc(60, 60, 10, 0, 2*Math.PI);
   ctx.fill();
```

JS (continue)

```
// draw a text
   ctx.fillStyle = 'purple';
   ctx.font = '20px Arial';
   ctx.fillText("你好", 60, 20);
}
```

Introduction to drawing/animating,

Azzeddine RIGAT

Introduction to drawing

Detail explanation:

- We use a function (`window.onload = function init(){`) called after the page is loaded (we say "after the DOM is ready"), so that the querySelector at (`canvas= document.querySelector('#myCanvas');`)will return the canvas. If the page was not completely loaded and if this code had been run before it had finished loading, the canvas value would have been "**undefined**".
- Once we have the canvas, we request a "graphic context" (`ctx = canvas.getContext('2d');`). This is a variable for 2D or 3D drawing on a canvas (in our case: 2D!) that we will use for drawing or setting colors, line widths, text fonts, etc.
- Then we can draw. Here we show only a few things you can do with the canvas API, but believe me, you can do much more (draw images, gradients, textures, etc.)! At (`ctx.fillRect(10, 10, 30, 30);`), we draw a filled rectangle. Parameters are the x and y coordinates of the **top left corner** (x goes to the right, y to the bottom of your screen), and the width and the height of the rectangle. At (`ctx.fillStyle = 'red';`), we used the **fillStyle** property of the context to set the color of filled shapes. This means: "now, all filled shapes you are going to draw will be in red!". It's like a global setting.

Introduction to drawing/animating,

Azzeddine RIGAT

Introduction to drawing

- Lines (`ctx.strokeStyle = 'green'; ctx.lineWidth = 4; ctx.strokeRect(100, 40, 40, 40);`) draw a green **wireframe rectangle**, with a line width equal to **4 pixels**. Notice the use of **`"stroke"`** instead of **`"fill"`** in the property name **`strokeStyle/fillStyle`** and in the context method for drawing a rectangle **`strokeRect/fillRect`**.
- Lines (`ctx.beginPath(); ctx.arc(60, 60, 10, 0, 2*Math.PI); ctx.fill();`) draw a *filled circle*. The syntax is a bit different as circles are parts of a *"path"*. Just keep in mind for now that before drawing a circle you need to call **`beginPath()`**. The call to **`arc(x, y, radius, start_angle, end_angle)`** does not draw the circle, it defines it. The next instruction **ctx.fill()** will draw all shapes that have been defined since a new path began, as filled shapes. Calling ctx.stroke() here, instead of ctx.fill() would have drawn a wireframe circle instead of a filled one. Also note that the filled circle is red even if we did not specify the color. Remember that we set *ctx.fillStyle = 'red'*. Unless we change this, all filled shapes will be red.
- Lines (`ctx.fillStyle = 'purple'; ctx.font = '20px Arial'; ctx.fillText("你好", 60, 20);`) draw a filled text. The call to **`filltext(message, x, y)`** draws a filled text at the x,y position; this time in purple as we called *ctx.fillStyle='purple'* before calling fillText(...)

Introduction to drawing/animating,

Azzeddine RIGAT

Introduction to drawing

Assignment:

Write down a script that displays the output shown below

Introduction to drawing/animating,

Azzeddine
RIGAT

Introduction to drawing

The second example emphasize on the functions that **save** and **restore** the context before drawing, check **example.II.2.6.02.Canvas**.

Detail explanation:

This time we've written two functions for a cleaner code: one function that draws a filled rectangle with a given color, and one function that draws a filled circle, with a given color.

The values for **`x, y, width, height, radius, color`** can be passed as parameters to these functions.

When a function changes anything to the "global context": filled or stroke color, line width, or the position of the coordinate system (located by default in 0, 0, at the top left of the canvas), then it is good practice to save this context at the beginning of the function, with a call to ctx.save(), and to restore it at the end of the function, with a call to ctx.restore(). **In this way, any change to the "global context" won't have any effect outside of the function**.

Introduction to drawing/animating,

Azzeddine
RIGAT

Introduction to drawing

Assignment:

- Check it by yourself, if you comment all the ctx.save() and ctx.store(), and ctx.fillStyle for the circle you can notice that the canvas keep working with the the first declared color that is rectangle color; red.
- Then check to uncomment the ctx.save() and ctx.store() of the rectangle you can notice that the red color stay just between the save() and the store, and when the canvas arrive to color the circle it got the black color, which is the default color of the canvas API.

We used also **`ctx.translate(x, y)`** in order to move the rectangle and the circle (look, they have been drawn at x=0, y=0, but as we translate the origin of the coordinate system with ctx.translate, the shapes are located in x, y on in the canvas). This is also a good practice: indeed, if we add more shapes (like eyes in the rectangle, in order to draw a monster), using coordinates relative to 0, 0, the whole set of shapes will be translated by the call to ctx.translate(x, y). This will make it easier to draw characters, monsters, etc. as we will see in a third example.

Introduction to drawing/animating,

Azzeddine RIGAT

Introduction to drawing

In the third example **example.II.2.6.03.Canvas**: we draw a face instead of a simple rectangle or circle. This is where you reap the benefits of your good habits of saving/restoring the context and using ctx.translate(x, y)

Detail explanation:

In this small example, we used the context object to draw a face using the default color (black), wireframe and filled modes:

- ctx.fillRect(x, y, width, height): draws a rectangle whose top left corner is at (x, y) and whose size is specified by the width and height parameters; and both outlined by, and filled with, the default color.
- ctx.strokeRect(x, y, width, height): same but in wireframe mode.

Introduction to drawing/animating,

Azzeddine
RIGAT

Introduction to drawing

- Note that **we use ctx.translate(x, y) to make it easier to move the face around**. So, all the drawing instructions are coded as if the face was in (0, 0), at the top left corner of the canvas (`ctx.strokeRect(0, 0, 100, 100);`). We draw the body outline with a rectangle starting from (0, 0). Calling **context.translate** "changes the coordinate system" by **moving the "old (0, 0)" to (x, y)** and keeping other coordinates in the same position relative to the origin.
- `drawFace(10, 10);` we call the drawFace function with (10, 10) as parameters, which will cause the original coordinate system to be translated by (10, 10).
- And if we change the coordinate system (this is what the call to ctx.translate(...) does) in a function, it is good practice to always save the previous context at the beginning of the function and restore it at the end of the function.

Assignment: Change the position of the face to the left bottom of the canvas.

Introduction to drawing/animating,

Azzeddine RIGAT

Animating

Animating: A typical animation loop will do the following at regular intervals:

1. Clear the canvas
2. Draw graphic objects / shapes
3. Move graphic shapes / objects
4. Go to step 1

Optional steps can be:

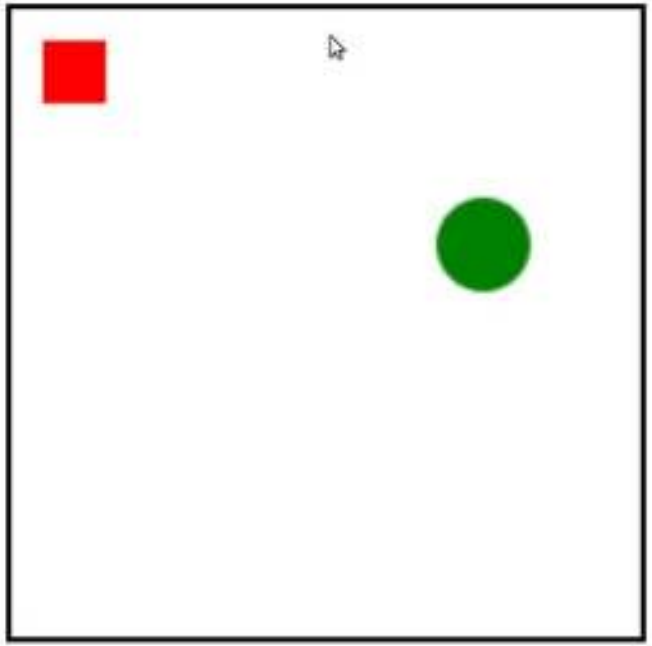
- Look at the keyboard / mouse / gamepad if we need to do something according to their status (i.e. if the left arrow is pressed: move the player to the left)
- Test collisions: the player collided with an enemy, remove one life
- Test game states: if there are no more lives, then go to the "game over" state and display a "game over" menu.
- Etc.

Introduction to drawing/animating,

Azzeddine RIGAT

Animating

Check **example.II.2.6.07.Canvas**, which animate a ball that bounces on the sides of the canvas (walls)



Introduction to drawing/animating,

Azzeddine RIGAT

Animating

Detail explanation: This time we've used "simple objects" for the circle and the rectangles, and we've called them "player" and "ball":

```
7   var  ball = {
8   y: 100,
9   x: 100,
10  radius: 15,
11  color: 'green',
12  speedX: 2,
13  speedY: 1
14  }
15
16  var  rectangle = {
17      y: 10,
18      x: 10,
19      width: 20,
20      height: 20,
21      color: 'red'
22      }
```

With this syntax, it's easier to manipulate "the x pos of the ball" - you just have to use ball.x. we added two properties to the ball object: speedX and speedY. Their value is the number of pixels that will be added to the current ball.x and ball.y position, at each frame of animation.

Introduction to drawing/animating,

Azzeddine RIGAT

Animating

Let's look at the animation loop:

```
function animateFilledCircle(){

    // clear the canvas
    ctx.clearRect(0, 0, w, h);

    // draw teh ball and the rectangle
    drawFilledRectangle(rectangle);
    drawFilledCircle(ball);

    //animate the ball that is bouncing all over the walls
    moveBall(ball);

    // cursively call this function 60 times per samcond
    requestAnimationFrame(animateFilledCircle);
}
```

Now, let's decompose the animation loop in some external functions to make it more readable. At each frame of animation, we will clear the canvas, draw the player as a rectangle, draw the ball as a circle, and move the ball.

You can take a look at the new versions of drawFilledRectangle that now take only one parameter named rectangle, instead of x, y, width, height and a color. We've only changed a few things in its code (changed x to r.x, y to r.y, color to r.color etc.)

Introduction to drawing/animating,

Azzeddine RIGAT

Animating

Let's look at the moveBall function:

```
59  function moveBall(circle){
60      //Check for collision with vertical walls
61      if(circle.x + circle.radius > w || circle.x - circle.radius < 0){
62          direction[0] *= -1;
63      }
64
65      //Check for collision with horizontal walls
66      if(circle.y + circle.radius > h || circle.y - circle.radius < 0){
67          direction[1] *= -1;
68      }
69      //update the ball position
70      circle.x += circle.speedX * direction[0];
71      circle.y += circle.speedY * direction[1];
72
73  }
```

This function is called 60 times per second. So, 60 times per second we modify the circle.x and circle.y positions of the ball passed as parameter by adding to them the b.speedX and b.speedY property values.

Notice that we call moveBall(ball) from mainLoop. In the moveBall function, the ball passed as a parameter becomes the circle parameter. So when we change the circle.x value inside the function, we are in reality changing the x value of the global object ball!

Ok, and at lines 61 and 66 we test if the ball hits a vertical or horizontal wall, respectively.

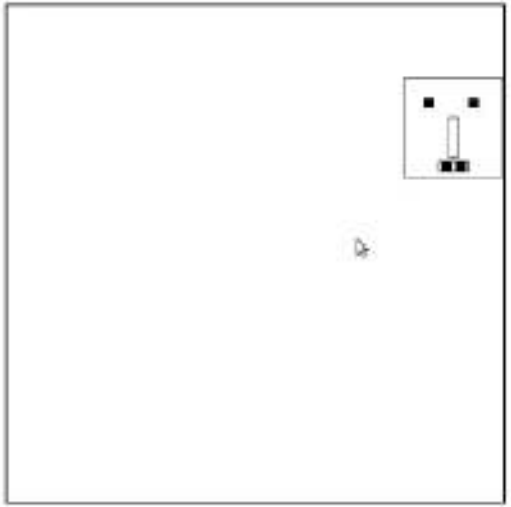
Introduction to drawing/animating,

Azzeddine RIGAT

Animating

Let's do some animation, check **example.II.2.6.04.Canvas**. Which animates the last face drawing by moving horizontally. The trick is to write a function, and at the end of this function, to ask the browser to call it again 60 times per second if possible on line 54(`requestAnimationFrame(moveFace);`).

Assignment: Write down a script that displays the output shown below;let the face move along the edges of the canvas in the clockwise direction

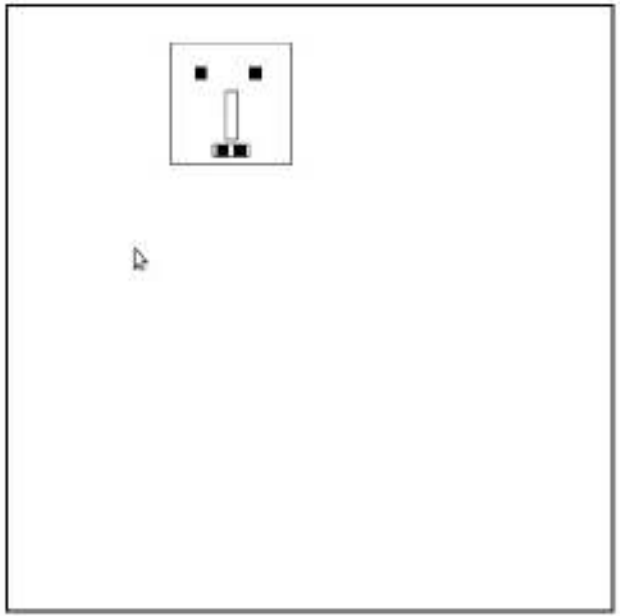


Introduction to drawing/animating,

Azzeddine
RIGAT

Animating

Assignment: Write down a script that displays the output shown below; let the face move randomly for every step of face between the edges of the canvas.



Introduction to drawing/animating,

Azzeddine RIGAT

Animating multiple objects

Check **example.II.2.6.08.Canvas**, which animate three balls that bounce on the sides of the canvas (walls). Extract of the source code: the main function; animateFilledCircles.

```
51  function animateFilledCircles(){
52      // clear the canvas
53      ctx.clearRect(0, 0, w, h);
54
55      drawFilledRectangle(rectangle);
56      //draw the balls
57      drawFilledCircle(ball1);
58      drawFilledCircle(ball2);
59      drawFilledCircle(ball3);
60
61      // animate the balls bouncing all over the walls
62      moveBall(ball1);
63      moveBall(ball2);
64      moveBall(ball3);
65
66      requestAnimationFrame(animateFilledCircles);
67  }
```

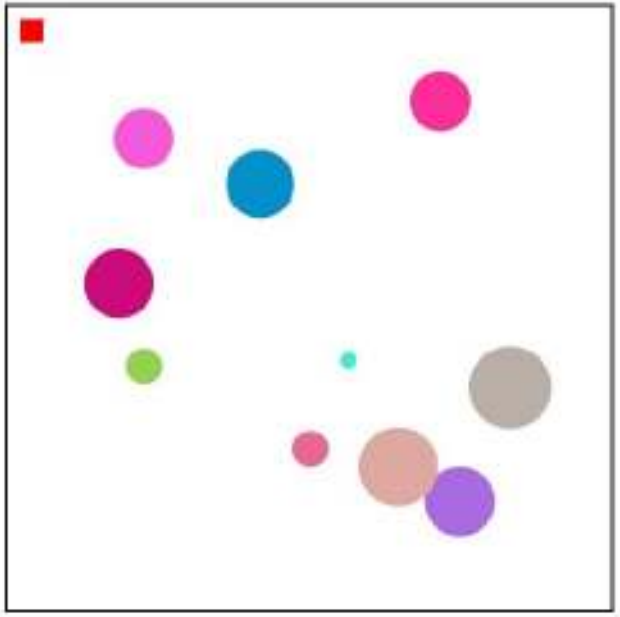
And what if we have **100 balls**? We're not going to copy and paste the lines that draw and move the balls 100 times!

Introduction to drawing/animating,

Azzeddine
RIGAT

Animating multiple objects

Assignment: Using arrays and loops for creating any number of balls, for animating and moving any number of balls!



Introduction to drawing/animating,

Azzeddine RIGAT

Animating multiple objects

Detail explanation: Let's loLet's look at the new functions we've added:

```
28  function createBalls(n){
29      //empty array
30      var ball_array =[];
31      // create n balls
32      for (var i=0; i<n; i++){
33          var  ball = {
34              y: w/2,
35              x: h/2,
36              radius: Math.random() * 30 + 5,// radius between 5 and 35
37              color: getARandomColor(),
38              speedX: -5 + Math.random() * 10, // speedX between -5 and 5
39              speedY: -5 + Math.random() * 10 // speedY between -5 and 5
40          }
41          // add ball to the array balls
42          ball_array.push(ball);
43      }
44      return ball_array;
45  }
```

createBalls(numberOfBalls), returns an array of balls:

- Line 30: we declare an empty array that will contain the balls,
- Lines 33-40: we create a new ball object with random values. Note the use of Math.random(), a predefined JavaScript function that returns a decimal value between 0 and 1. We call another function named getARandomColor() that returns a color taken randomly.
- Line 42: we add the newly created ball b to the array,
- Line 44: we return the array to the caller.

Introduction to drawing/animating,

Azzeddine RIGAT

Animating multiple objects

```
110  function getARandomColor(){
111      var r = Math.random() * 256;
112      var g = Math.random() * 256;
113      var b = Math.random() * 256;
114
115      return 'rgb(' + r + ',' + g + ',' + b + ')';
116  }
```

- Lines 111-113: we generate a random number between 0 and 255.
- Line 115: we use the rgb JavaScript function that generate a color using the the three primitive colors red, green, and blue that have a range between 0 and 255.

Introduction to drawing/animating,

Azzeddine RIGAT

Animating multiple objects

```
95   //draw the balls
96   function drawFilledCircles(balls){
97       balls.forEach( ball => {
98           drawFilledCircle(ball);
99       });
100  }
```

```
102  //draw the balls
103  function moveBalls(balls){
104      balls.forEach(ball => {
105          moveBall(ball)
106      });
107  }
```

These two functions use an iterator on the array of balls.

This script is equivalent the second left side function:

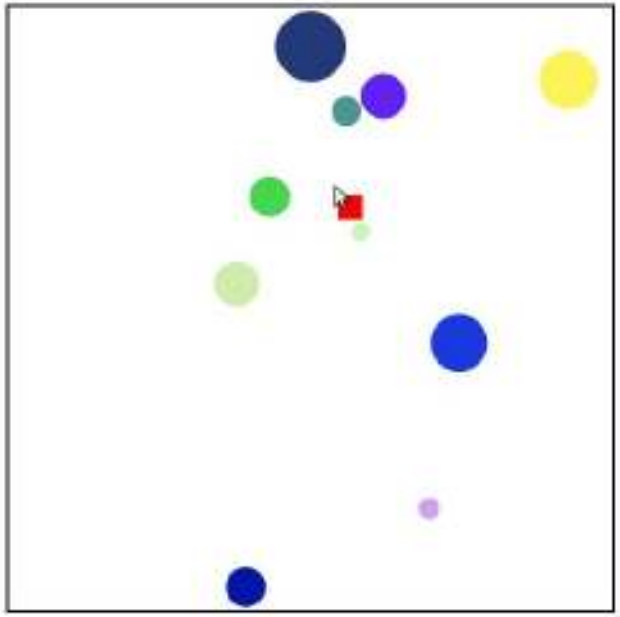
```
function moveBalls(balls){
   for(var i = 0; i< balls.length; i++){
       moveBall(balls[i]);
   }
}
```

Introduction to drawing/animating,

Azzeddine RIGAT

Moving a rectangle with the mouse

Assignment: This time, add a **`mousemove`** event listener to the canvas in the init function, and reused the trick that you saw in the mouse events section to get the correct mouse position:



Introduction to drawing/animating,

Azzeddine RIGAT

Moving a rectangle with the mouse

Detail explanation: Let's look at the new functionalities we've added:

1. We need to add inside the init function:

```
// add listener to the canvas
canvas.addEventListener('mousemove', function(event){
    processMouseMove(event);});
// get canvas rect
canvas_rect = canvas.getBoundingClientRect();
```

2. Then, we define processMouseMove function: here we are going to update the position of the rectangle everytime we call the mousemove event.

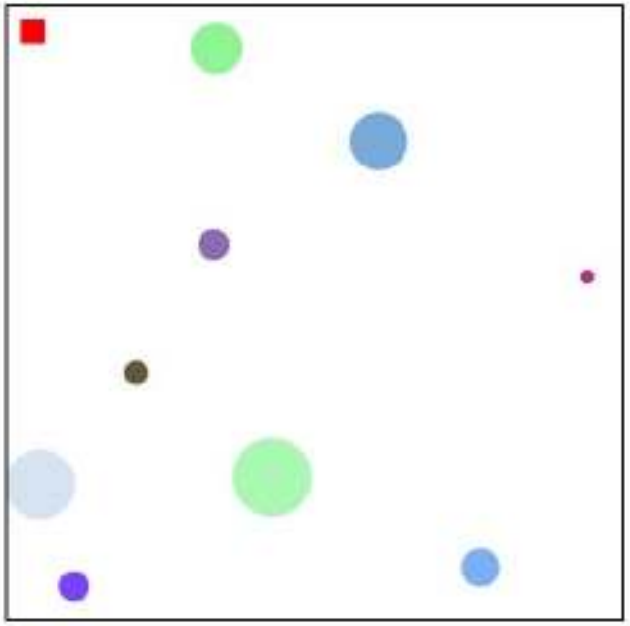
```
function processMouseMove(event){
   rectangle.x = event.clientX - canvas_rect.left;
   rectangle.y = event.clientY - canvas_rect.top;}
```

Introduction to drawing/animating,

Azzeddine RIGAT

Making it a game! Adding collision detection

Assignment: This time, add a **`mousemove`** event listener to the canvas in the init function, and reused the trick that you saw in the mouse events section to get the correct mouse position:

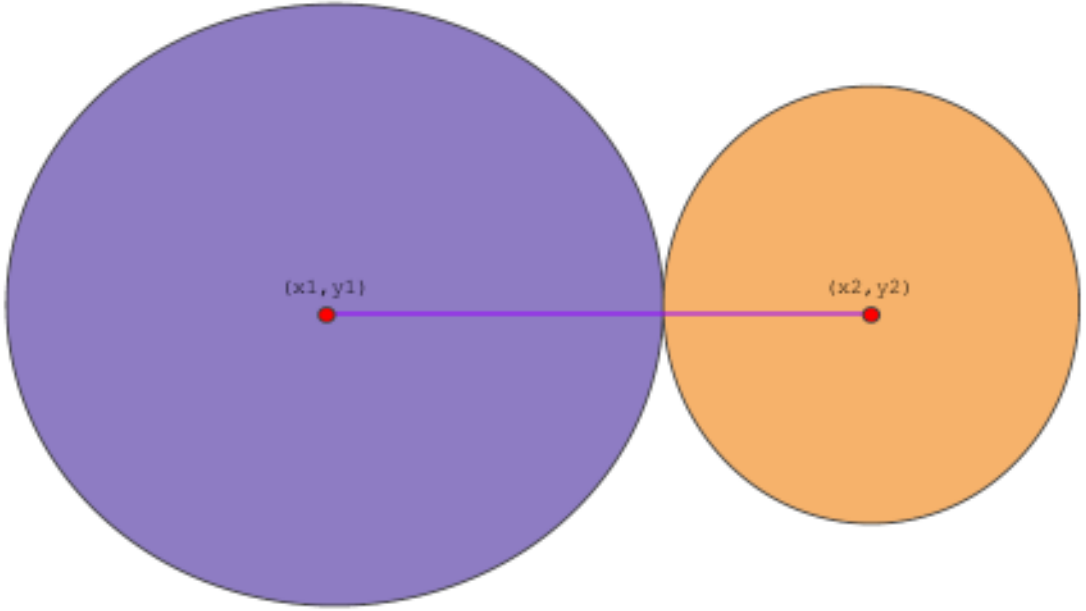


Introduction to drawing/animating,

Azzeddine RIGAT

Making it a game! Adding collision detection

How do we detect collisions between two circles?



Suppose that

- C1 with center at (x1,y1) and radius r1;
- C2 with center at (x2,y2) and radius r2.

Introduction to drawing/animating,

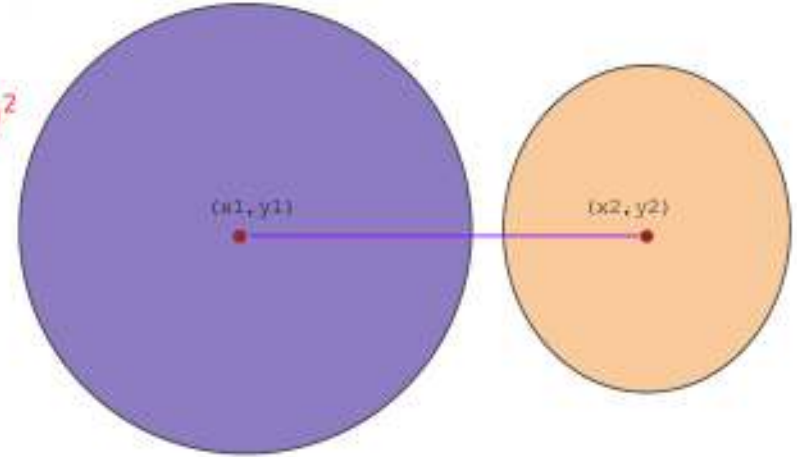
Azzeddine RIGAT

Making it a game! Adding collision detection

Imagine there is a line running between those two center points. The distance from the center points to the edge of either circle is, by definition, equal to their respective radius. So:

if the edges of the circles touch, the distance between the centers is r1+r2;
any greater distance and the circles don't touch or collide; and
any less and then do collide.

So you can detect collision if: $(x_2-x_1)^2 + (y_1-y_2)^2 <= (r_1+r_2)^2$

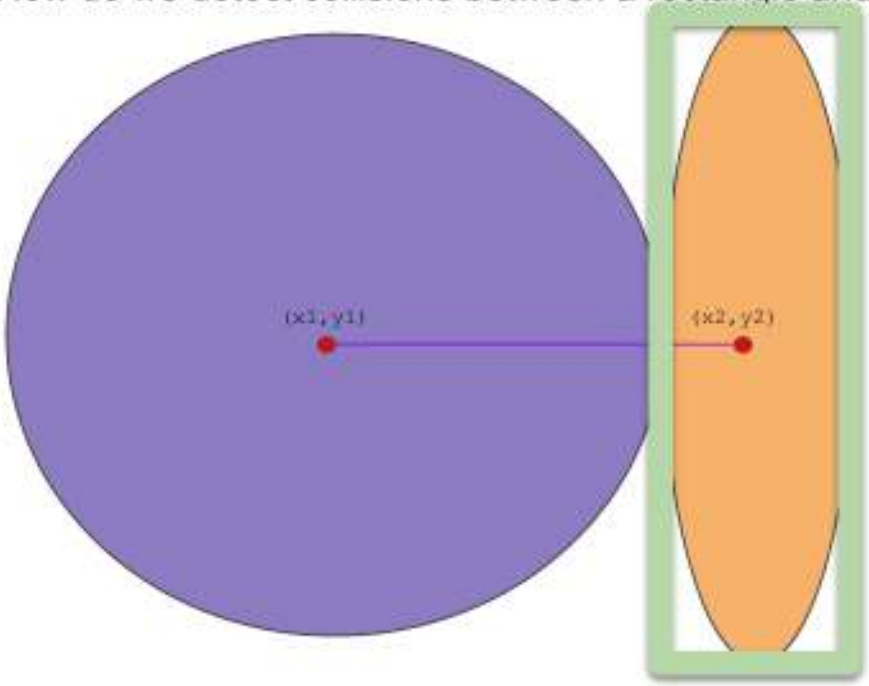


Introduction to drawing/animating,

Azzeddine RIGAT

Making it a game! Adding collision detection

How do we detect collisions between a rectangle and a circle?



In case, we try to replace the rectangle with an ellipse, it won't be easy because of its radius.

So, we are going to find the the different cases where a rectangle intersect with a circle.

Introduction to drawing/animating,

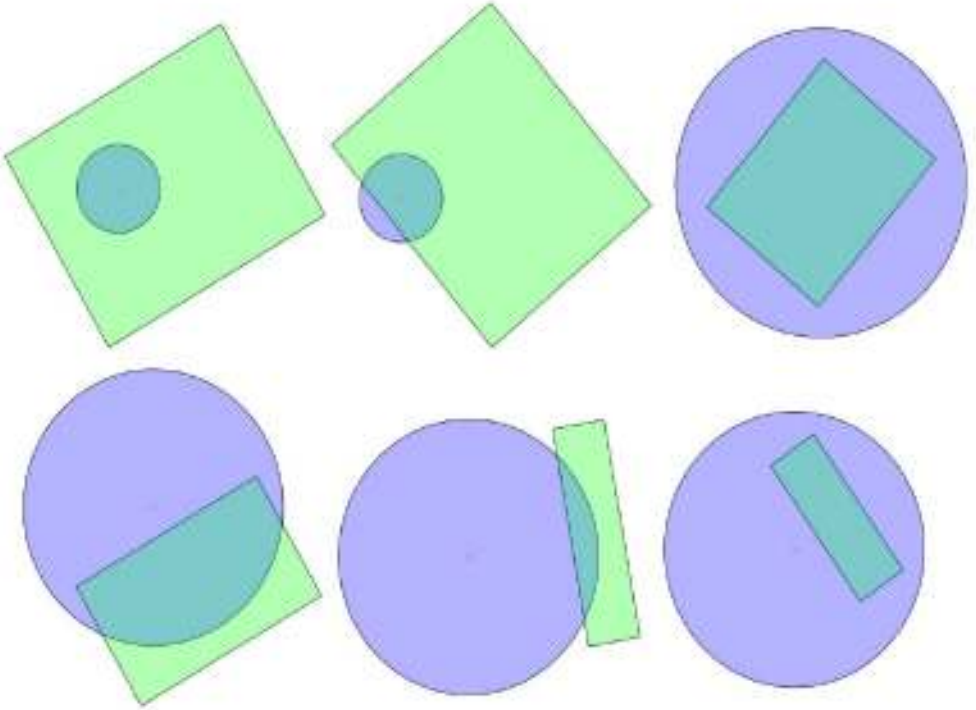
Azzeddine RIGAT

Making it a game! Adding collision detection

Actually, there are only two cases when the circle intersects with the rectangle:

- Either the circle's centre lies inside the rectangle, or
- One of the edges of the rectangle has a point in the circle.

Note that this does not require the rectangle to be axis-parallel.



Introduction to drawing/animating,

Azzeddine RIGAT

Making it a game! Adding collision detection

JS

```
// Collisions between rectangle and cicles
function circleRectOverlap(x0, y0, w0, h0, cx, cy, r){
   // by this we assume that circle's center is inside the rect
   var testX = cx, testY= cy;
   // Then we we are going to check the position of the
   // circle's center compared with the edges of the of
   // the rectangle
   if (x0 + w0 < testX) testX = x0 + w0;
      else if (testX < x0) testX = x0;

   if (testY > y0 + h0) testY = y0 +h0;
      else if (y0 > testY) testY = y0;

   return (((testX - cx)*(testX - cx) + (testY - cy)*(testY - cy))<r*r);
}
```

Introduction to drawing/animating,

Azzeddine RIGAT

Making it a game! Adding collision detection

JS

```
//check collisions
function checkCollision(ball, index){
       if(circleRectOverlap(rectangle.x, rectangle.y, rectangle.width,
           rectangle.height, ball.x, ball.y, ball.radius)){
               // We remove the element located at index
               // from the balls array
               // splice: first parameter = starting index
               // second parameter = number of elements to remove
               balls.splice(index, 1);
           }
}
```

This time we've added into the loop a collision test between the player and the balls. If the player hits a ball, it's removed from the ball array. We did this test in the moveBalls function.

Introduction to drawing/animating,

Azzeddine RIGAT

Adding input fields to parameterize the game

Assignment: Let's use some other techniques that we've learnt during this course: some input fields: sliders, color chooser, number chooser, and use the DOM API to handle them.

We will use these input fields to indicate the number of balls we want, the max speed we would like, the color and size of the player, etc.

Introduction to drawing/animating,

Azzeddine RIGAT

Adding input fields to parameterize the game

Detail explanation:

In the HTML file, we use onchange inside the select tag to capture the event of changing the speed:

```
select id='enemyColor' onchange='choosenEnemyColor(this.value)'>
```

For simplicity, if the game is still on each time we change the number of balls in the game, we restart the game.

Introduction to drawing/animating,

Azzeddine RIGAT

Adding input fields to parameterize the game

Optional projects:

- The game is not completely finished, as you may have noticed :-) So, try to make "**levels**": when all good balls have been eaten, let's restart automatically, but this time with one more ball in the initial set!
- Include a wizard for parameterize your game; choose the number of balls first,...., then click on a button to start your game.
- Display the level number on the right,
- Try to use a global variable "gameState" that can be equal to "gameRunning" or to "displayGameOverMenu". Use it in the game loop with a switch statement, to display a game over menu when the player hits a certain number of bad balls (say three bad balls eaten and you're done!)
- When, in the game over menu, listen to keydown events on the canvas and suggest pressing space to go to restart the game.
- Add other input fields to further customize the game.

Introduction to drawing/animating,

Azzeddine RIGAT

Adding input fields to parameterize the game

- Draw a small face instead of a red square for the player, and do the same thing with the balls - make some improvements!
- Make ball movements different depending on their color (random direction that changes along the course of a ball, zigzag movements, etc.)
- Make the balls change their size during the animation.
- Make your rectangle fire bullets to destroy enemy balls :-)
- Use images instead of drawing.