

# [Lesson 11]

Roi Yehoshua 2018

## [ What we learnt last time? ]

- Working with DOM
- DOM Tree
- Searching nodes

## [Our targets for today]

- Working with DOM
- Inserting, moving, removing and cloning nodes
- Attributes
- Page Geometry

# [Custom DOM Properties]

- DOM nodes are regular JavaScript objects
- We can add our own properties and methods to them
- For instance, let's create a new property in document.body:

```
document.body.myData = { name: 'Ceaser', title:
    'Emperor'
};
alert(document.body.myData.title); // Emperor
```

- We can also modify built-in prototypes like Element.prototype, and add new methods to all elements:

```
Element.prototype.sayHi = function () { alert(`Hello, I'm
    ${this.tagName}`);
};

document.documentElement.sayHi(); // Hello, I'm HTML
document.body.sayHi(); // Hello, I'm BODY
```

# [HTML Attributes]

- When the browser loads the page, it “parses” HTML text and generates DOM objects from it
- For element nodes most standard HTML attributes automatically become properties of their corresponding DOM objects
- But the attribute-property mapping is not one-to-one!
  - For example, HTML attribute values are always strings while DOM properties are typed
  - If an HTML attribute is non-standard, there won't be DOM-property for it
- All HTML attributes are accessible using following methods:
  - **elem.hasAttribute(name)** – checks for existence
  - **elem.getAttribute(name)** – gets the value
  - **elem.setAttribute(name, value)** – sets the value
  - **elem.removeAttribute(name)** – removes the attribute

# [HTML Attributes]

→ Example for working with HTML attributes:

```
<div id="elem" about="Elephant"></div>
<script>
  alert(elem.getAttribute("about")); // 'Elephant', reading
  elem.setAttribute("Test", 123); // writing

  for (let attr of elem.attributes) { // list all attributes
    alert(`${attr.name} = ${attr.value}`);
  }
</script>
```

→ The HTML attribute may differ from its corresponding DOM property, for example the style attribute is a string, but the style property is an object:

```
<div id="div" style="color:red;font-size:120%">Hello</div>
<script>
  alert(div.getAttribute('style')); // color:red;font-size:120%
  alert(div.style); // [object CSSStyleDeclaration]
  alert(div.style.color); // red
</script>
```

# [HTML Custom Attributes]

- There is a possible problem with custom attributes
- What if we use a non-standard attribute for our purposes, and later the standard introduces it and makes it do something?
- To avoid conflicts, there exist **data-\*** attributes
- All attributes starting with “data-” are reserved for programmers’ use
- They are available in the **dataset** property:

```
<div id="elem" data-about="Elephants">  
  <script>  
    alert(elem.dataset.about); // Elephants  
  </script>  
</div>
```

- Multiword attributes like data-order-state become camel-cased: dataset.orderState

## [Exercise]

- Write the code to select the element with data-widget-name attribute from the document and to read the attribute's value

```
<!DOCTYPE html>

<html>
<body>
  <div data-widget-name="menu">Choose the genre</div>

  <script>
    /* your code */
  </script>
</body>
</html>
```



## [Element Style]

- The property `elem.style` is an object that corresponds to what's written in the "style" attribute
  - Setting `elem.style.width="100px"` works as if we had in the attribute `style="width:100px"`
- For multi-word property, camel casing is used:

```
background-color => elem.style.backgroundColor  z-index =>
elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

- For instance, the following script lets the user change the page's background color:

```
<script>
  document.body.style.backgroundColor = prompt('Background color?');
</script>
```

## [Mind the Units]

- CSS units must be provided in style values
- For instance, we should not set `elem.style.top` to 10, but rather to 10px

```
<div id="elem"> Hello world
</div>

<script>
  // doesn't work! elem.style.margin = 20;
  alert(elem.style.margin); // '' (empty string, the assignment is ignored)

  // now add the CSS unit (px) - and it works elem.style.margin = '20px';
  alert(elem.style.margin); // 20px

  alert(elem.style.marginTop); // 20px alert(elem.style.marginLeft); // 20px
</script>
```

# [Styles and Classes]

- There are generally two ways to style an element:
  - Create a class in CSS and add it: `<div class="...">`
  - Write properties directly into style: `<div style="...">`
- CSS is always the preferred way – not only for HTML, but in JavaScript as well
- We should only manipulate the style property if classes “can’t handle it”
- For instance, style is acceptable if we calculate coordinates of an element dynamically and want to set them from JavaScript, like this:

```
let top = /* complex calculations */; let left =  
/* complex calculations */; elem.style.left =  
left; // e.g '123px' elem.style.top = top; //  
e.g '456px'
```

## [className]

- In the ancient time, there was a limitation in JavaScript: a reserved word like "class" could not be an object property
- So the property "className" was introduced: **elem.className** corresponds to the "class" attribute
- For instance:

```
<h1 id="elem" class="header main"></h1>  
<script>  
    alert(elem.className); // header main  
</script>
```

- If we assign something to elem.className, it replaces the whole strings of classes

## [classList]

- Sometimes we only want to add/remove a single class
- There's another property for that: **elem.classList**
- Methods of classList:
  - **elem.classList.add/remove("class")** – adds/removes the class
  - **elem.classList.toggle("class")** – if the class exists, then removes it, otherwise adds it
  - **elem.classList.contains("class")** – returns true/false, checks for the given class
- For instance:

```
<h1 id="elem" class="header main"></h1>  
<script>  
  elem.classList.add("article");  
  alert(elem.className); // header main article  
</script>
```

# [Computed Styles]

- The **style** property operates only on the value of the "style" attribute, without any CSS cascade
- So we can't read anything that comes from CSS classes using `elem.style`
- For instance, here style doesn't see the margin:

```
<head>
  <style>
    body {
      color: red;
      margin: 5px
    }
  </style>
</head>
<body>
  The red text
  <script>
    alert(document.body.style.color); // empty
    alert(document.body.style.marginTop); // empty
  </script>
</body>
```

# [Computed Styles]

- The method `getComputedStyle(element)` returns an object with style properties, like `elem.style`, but with respect to all CSS classes:

```
<head>
  <style>
    body {
      color: red; margin: 5px
    }
  </style>
</head>
<body>
  The red text
  <script>
    let computedStyle = getComputedStyle(document.body);

    // now we can read the margin and the color from it
    alert(computedStyle.marginTop); // 5px
    alert(computedStyle.color); // rgb(255, 0, 0)
  </script>
</body>
```

# [Computed and Resolved Values]

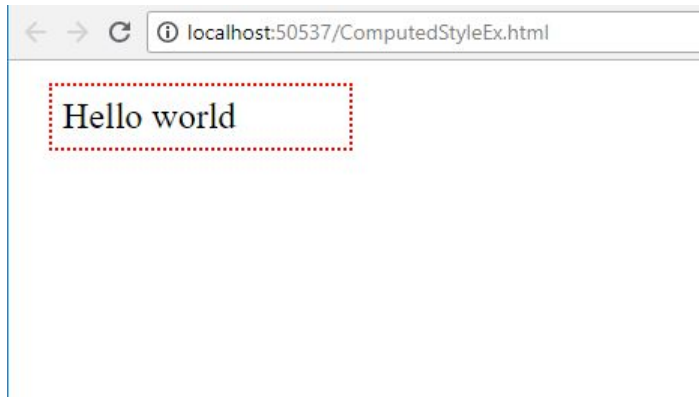
- There are two concepts in [CSS](#):
  - A *computed* style value is the value after all CSS rules and CSS inheritance is applied, as the result of the CSS cascade. It can look like height:1em or font-size:125%.
  - A *resolved* style value is the one finally applied to the element. Values like 1em or 125% are relative. The browser takes the computed value and makes all units fixed and absolute, for instance: height:20px or font-size:16px
- Originally, getComputedStyle() was created to get computed values, but it turned out that resolved values are much more convenient, and the standard changed
- Nowadays getComputedStyle() returns the resolved value of the property



## [Exercise (7)]

- Move the following div 20px to the right, by increasing its margin-left property
  - Hint: first use getComputedStyle() to get its current marginLeft value

```
<head>
  <style>
    #div1 {
      border: red dotted 1px;
      width: 20%;
      margin: 10px; padding: 5px;
    }
  </style>
</head>
<body>
  <div id="div1"> Hello world
</div>
</body>
```



# [Creating Elements]

→ `document.createElement(tag)` creates a new element with the given tag:

```
let div = document.createElement("div");
```

→ After that, we have a ready DOM element

→ To make the element show up, we need to insert it somewhere into document

→ There are several methods for inserting a node into a parent element

Method	Description
<code>parentElem.appendChild(<i>node</i>)</code>	appends <i>node</i> as the last child of <i>parentElem</i>
<code>parentElem.insertBefore(<i>node</i>, <i>nextSibling</i>)</code>	inserts <i>node</i> before <i>nextSibling</i> into <i>parentElem</i>
<code>parentElem.replaceChild(<i>node</i>, <i>oldChild</i>)</code>	replaces <i>oldChild</i> with <i>node</i> among children

# [Creating Elements]

→ The following example adds a new <li> to the end of <ol>:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  let newLi = document.createElement("li");
  newLi.innerHTML = "Hello, world!";

  list.appendChild(newLi);
</script>
```

1. 0  
2. 1  
3. 2  
4. Hello, world!

## [Creating Elements]

→ The following code inserts a new list item before the second <li>:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  let newLi = document.createElement("li");
  newLi.innerHTML = "Hello, world!";

  list.insertBefore(newLi, list.children[1]);
</script>
```

1. 0
2. Hello, world!
3. 1
4. 2

## [Insertion Methods]

→ There is another set of methods that provide more flexible insertions:

Method	Description
<code>node.append(...<i>nodes</i> or <i>strings</i>)</code>	appends <i>nodes</i> or <i>strings</i> at the end of node
<code>node.prepend(...<i>nodes</i> or <i>strings</i>)</code>	inserts <i>nodes</i> or <i>strings</i> into the beginning of node
<code>node.before(...<i>nodes</i> or <i>strings</i>)</code>	inserts <i>nodes</i> or <i>strings</i> before the node
<code>node.after(...<i>nodes</i> or <i>strings</i>)</code>	inserts <i>nodes</i> or <i>strings</i> after the node
<code>node.replaceWith(...<i>nodes</i> or <i>strings</i>)</code>	replaces node with the given <i>nodes</i> or <i>strings</i>

# [Insertion Methods]

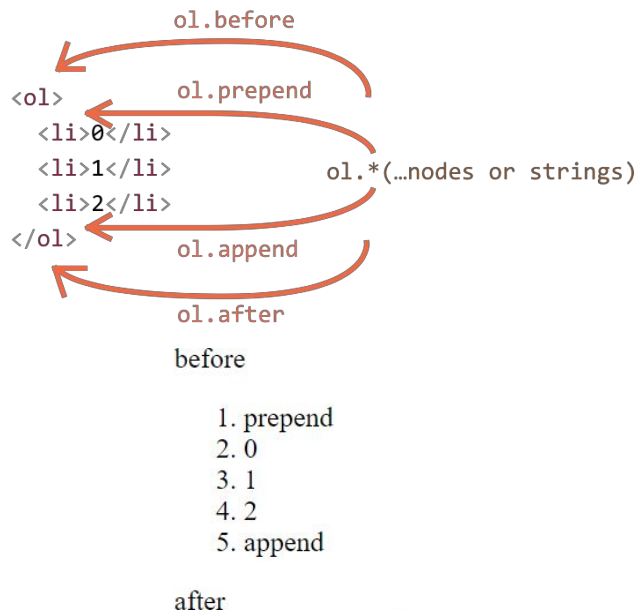
→ Here's an example of using these methods to add more items to a list and the text before/after it:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  list.before("before");
  list.after("after");

  let prepend = document.createElement("li");
  prepend.innerHTML = "prepend";
  list.prepend(prepend);

  let append = document.createElement("li");
  append.innerHTML = "append";
  list.append(append);
</script>
```



## [Cloning Nodes]

- Sometimes when we have a big element, it may be faster and simpler to clone it rather than create a new element
- **elem.cloneNode(true)** creates a “deep” clone of the element
  - with all attributes and subelements
- **elem.cloneNode(false)** creates a clone without child elements

# [Cloning Nodes]

→ An example of copying a <div> tag showing a message:

```
<head>
  <style>
    .alert {
      padding: 15px;
      border: 1px solid #d6e9c6; border-radius: 4px; color:
      #3c763d; background-color: #dff0d8;
    }
  </style>
</head>
<body>
  <div class="alert" id="div">
    <strong>Hi there!</strong> You've read an important message.
  </div>
  <script>
    let div2 = div.cloneNode(true); // clone the message
    div2.querySelector('strong').innerHTML = 'Bye there!'; //
    change the clone
    div.after(div2); // show the clone after the existing div
  </script>
</body>
```

Hi there! You've read an important message.

Bye there! You've read an important message.



# [Removal Methods]

→ To remove nodes, there are the following methods:

Method	Description
<code>parentElem.removeChild(<i>node</i>)</code>	removes <i>elem</i> from <i>parentElem</i> (assuming it's a child)
<code>node.remove()</code>	Removes the node from its place

→ The second method is much shorter. The first one exists for historical reasons.

→ For example, let's make our message disappear after a second:

```
<div class="alert" id="div">
  <strong>Hi there!</strong> You've read an important message.
</div>

<script>
  setTimeout(() => div.remove(), 1000);
</script>
```

## [Moving Nodes]

- If we want to *move* an element to another place – there's no need to remove it from the old one.
- All insertion methods automatically remove the node from the old place
- For instance, let's swap elements:

```
<div id="first">First</div>
<div id="second">Second</div>

<script>
  // no need to call remove
  second.after(first); // take #second and after it - insert
  #first
</script>
```

Second  
First

## [Exercise (8)]

- Create a function **clear(elem)** that removes everything from inside the element (but keeps the element itself)

```
<ol id="list">
  <li>Hello</li>
  <li>World</li>
</ol>

<script>
  function clear(elem) { /* your code */ }

  clear(list); // clears the list
</script>
```

## [Exercise (9)]

→ Write the code to insert the elements `<li>2</li>` and `<li>3</li>` between the two `<li>` here:

```
<ul id="ul">  
  <li id="one">1</li>  
  <li id="two">4</li>  
</ul>
```

## [Exercise (10)]

- Write a function **createCalendar**(elem, year, month)
- The call should create a calendar for the given year/month and put it inside elem
- The calendar should be a table, where a week is <tr>, and a day is <td>
- The table top should be <th> with weekday names
- For instance, createCalendar(cal, 2018, 6) should generate in element cal the following calendar:

SU	MO	TU	WE	TH	FR	SA
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

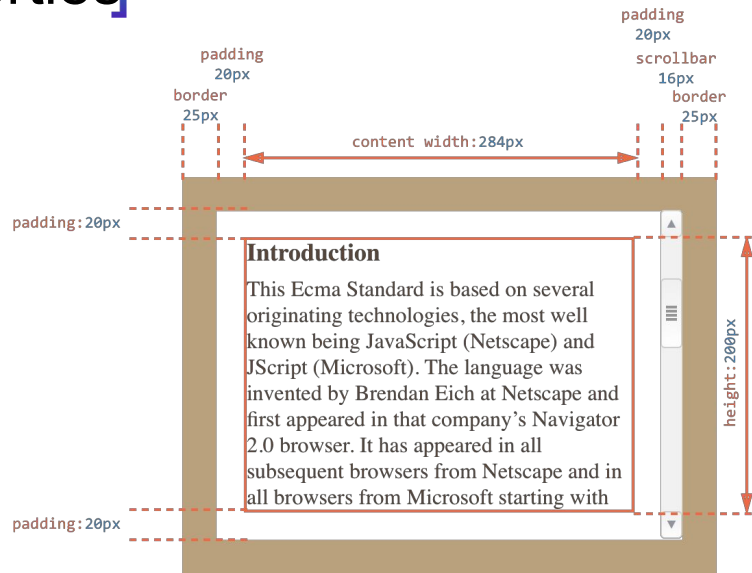
# [Element Geometry]

- There are many JavaScript properties that allow us to read information about element width, height and other geometry features
- We often need them when moving or positioning elements in JavaScript, to correctly calculate coordinates
- As a sample element to demonstrate properties we'll use the one given below:

```
#example {  
  width: 300px; height: 200px;  
  overflow: auto;  
  border: 25px solid #E8C48F;  
  padding: 20px;  
}  
  
<div id="example">  
  ...Text...  
</div>
```

# [Element Geometry Properties]

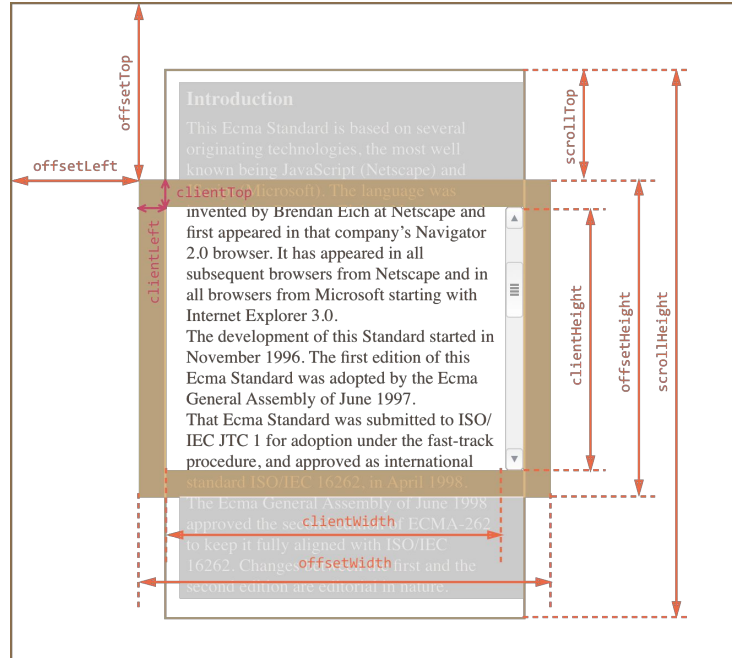
→ The element looks like this:



- Mind the scrollbar: most browsers reserve the space for it by taking it from the content, so if the scrollbar is 16px width (the width may vary between devices and browsers) then only  $300 - 16 = 284\text{px}$  remains for the content

# [Element Geometry Properties]

- Element properties that provide width, height and other geometry are always numbers. They are assumed to be in pixels.



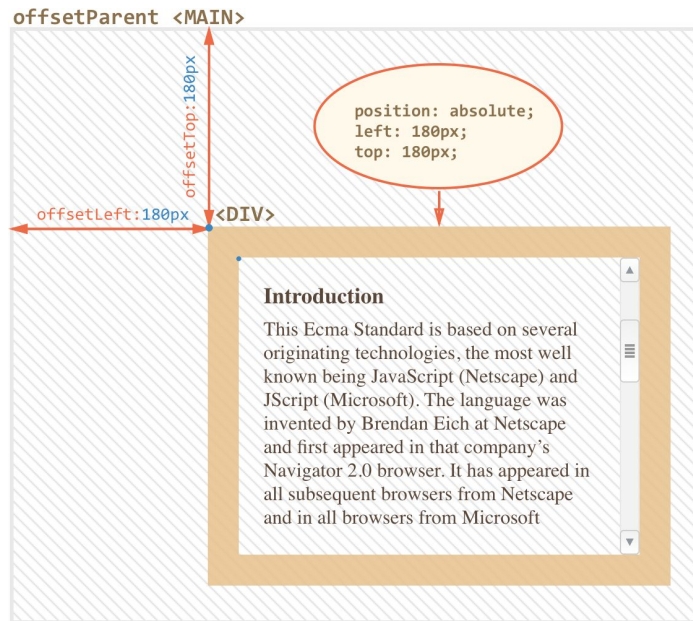


## [offsetParent, offsetLeft/Top]

- These are the “most outer” geometry properties
- The **offsetParent** is the nearest ancestor that is:
  - CSS-positioned (position is absolute, relative or fixed),
  - or <td>, <th>, <table>
  - or <body>
- **offsetLeft/offsetTop** provide x/y coordinates relative to the parent’s left-upper corner
- In the example below inner <div> has <main> as offsetParent and offsetLeft/offsetTop shifts from its left-upper corner (180)

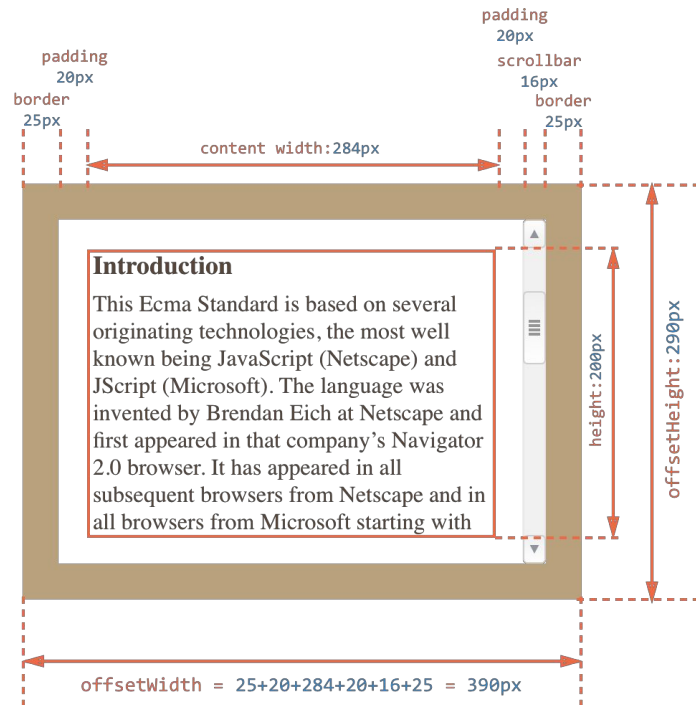
# [offsetParent, offsetLeft/Top]

```
<main style="position: relative" id="main">
  <article>
    <div id="example" style="position: absolute; left:
180px; top: 180px">
      Text...
    </div>
  </article>
</main>
<script>
  alert(example.offsetParent.id); // main
  alert(example.offsetLeft); // 180 (note: a number, not a
string "180px") alert(example.offsetTop); // 180
</script>
```



# [offsetWidth/Height]

- The properties **offsetwidth** and **offsetheight** provide the “outer” width/height of the element, i.e., its full size including borders
- The **offsetWidth** is calculated as inner CSS- width (300px) plus paddings (2 \* 20px) and borders (2 \* 25px)



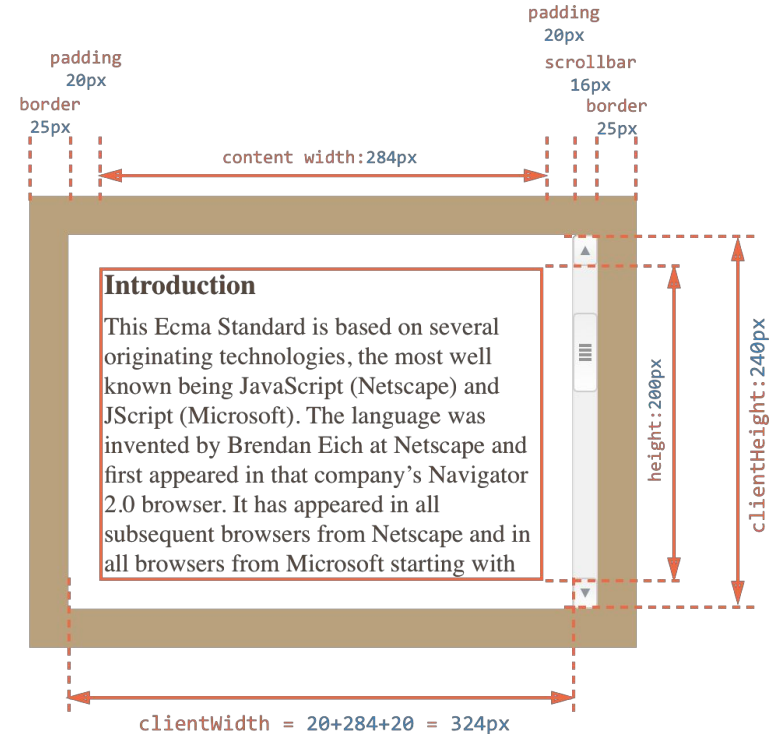
## [clientTop/Left]

- The properties **clientTop** and **clientLeft** measure the border size
- In our example:
  - $\text{clientLeft} = 25 - \text{left border width}$
  - $\text{clientTop} = 25 - \text{top border width}$
- To be precise – they measure the relative coordinates of the inner side from the outer side
- When the document is right-to-left, the scrollbar is then on the left, and then **clientLeft** also includes the scrollbar width
  - e.g., in the page on the right the **clientLeft** also includes the scrollbar width:  $25 + 16 = 41$



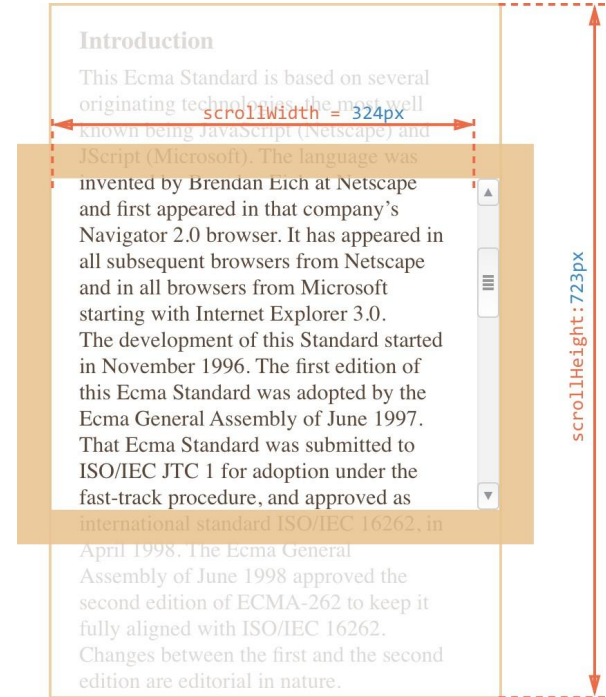
# [clientWidth/Height]

- These properties provide the size of the area inside the element borders
- They include the content width together with paddings, but without the scrollbar
- If there are no paddings, then clientWidth/Height is exactly the content area, inside the borders and the scrollbar (if any)



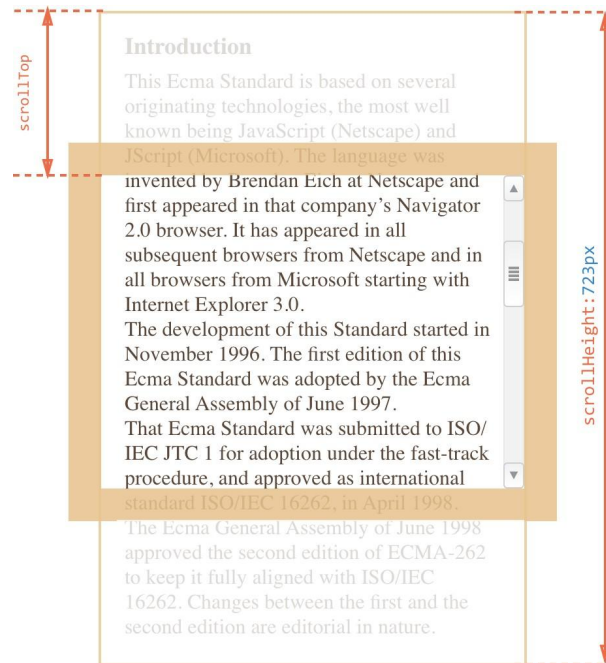
# [scrollWidth/Height]

- Properties `clientWidth/clientHeight` only account for the visible part of the element.
  - Properties **`scrollWidth/scrollHeight`** also include the scrolled out (hidden) parts
- On the picture on the right:
  - `scrollHeight = 723` – is the full inner height of the content area including the scrolled out parts
  - `scrollWidth = 324` – is the full inner width, here we have no horizontal scroll, so it equals `clientWidth`
- We can use these properties to expand the element wide to its full width/height:
  - `element.style.height = `${element.scrollHeight}px`;`



## [scrollTop/Top]

- Properties **scrollTop/scrollTop** are the width/height of the hidden, scrolled out part of the element
- On the picture on the right we can see **scrollHeight** and **scrollTop** for a block with a vertical scroll
- **scrollLeft/scrollTop** can be changed, and the browser will scroll the element



# [Don't Take Width/Height from CSS]

- We've just covered geometry properties of DOM elements
- They are normally used to get widths, heights and calculate distances
- But as we know, we can read CSS-height and width using `getComputedStyle()`
- Why should we use geometry properties instead? There are a few reasons:
  - First, CSS width/height depend on another property: box-sizing that defines “what is” CSS width and height. A change in box-sizing for CSS purposes may break such JavaScript.
  - Second, CSS width/height may be auto, for instance for an inline element:

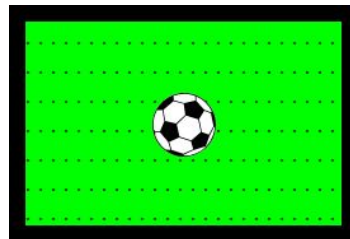
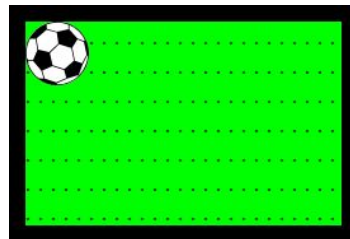
```
<span id="elem">Hello!</span>
<script>
    alert(getComputedStyle(elem).width); // auto
</script>
```

- Third, `clientWidth/clientHeight` take the scrollbar size into account, while in some browsers (e.g., Firefox) `getComputedStyle(elem).width` returns the CSS width (ignore the scrollbar)



## [Exercise (11)]

- What are the coordinates of the field center?
- Calculate them and use to place the ball into the center of the field
- The element should be moved by JavaScript, not CSS
- The code should work with any ball size (10, 20, 30 pixels) and any field size, not be bound to the given values
- Start with the HTML page on next slide



## [Exercise (11)]

```
<!DOCTYPE HTML>
<html>
<head>
  <style>
    #field {
      width: 200px;
      border: 10px groove black; background-color: #00FF00; position: relative;
    }

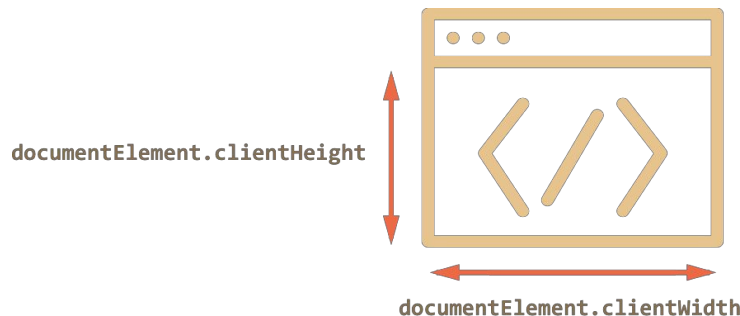
    #ball {
      position: absolute;
    }
  </style>
</head>
<body>
  <div id="field">
    
    . . . . .
    . . . . .
    . . . . .
    . . . . .
    . . .
  </div>
</body>
</html>
```

# [Window Geometry]

- From the DOM point of view, `document.documentElement` corresponds the root `<html>` tag, and has geometry properties described previously
- In some cases we can use it, but there are additional methods and peculiarities important enough to consider

## [Width/Height of the Window]

- Properties `clientWidth/clientHeight` of `document.documentElement` provide the width/height of the window, ignoring any scrollbars



- Browsers also support properties `window.innerWidth/innerHeight`, which provide the full window width/height (including scrollbars)
- Typically we need the available window width to draw or position something, i.e., inside scrollbars if there are any

## [Width/Height of the Document]

- Due to browser inconsistencies, to get the width/height of the whole document, with the scrolled out part, we should use the following code:

```
let scrollHeight = Math.max(  
    document.body.scrollHeight,  
    document.documentElement.scrollHeight,  
    document.body.offsetHeight,  
    document.documentElement.offsetHeight,  
    document.body.clientHeight,  
    document.documentElement.clientHeight  
);  
  
alert("Full document height, with scrolled out part: " + scrollHeight);
```

- To get the current scroll state, use the special properties `window.pageXOffset/pageYOffset`:

```
alert("Current scroll from the top: " + window.pageYOffset);  
alert("Current scroll from the left: " + window.pageXOffset);
```

## [Changing the Current Scroll ]

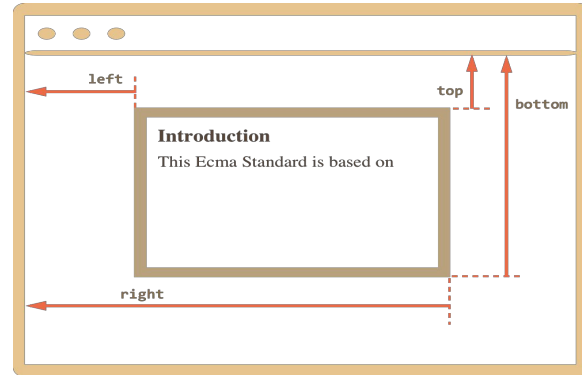
- To change the current scroll use one of the following methods:
  - **window.scrollTo(pageX,pageY)** – absolute coordinates
    - For instance, to scroll to the very beginning of the page, can use `scrollTo(0,0)`
  - **window.scrollBy(x,y)** – scroll relative the current place
    - For instance, `scrollBy(0,10)` scrolls the page 10px down
  - **elem.scrollIntoView(top)** – scroll to make **elem** visible (align with the top/bottom of the window)

# [Coordinates]

- To move elements around we should be familiar with coordinates
- Most JavaScript methods deal with one of two coordinate systems:
  - Relative to the window (or another viewport) top/left
  - Relative to the document top/left
- It's important to understand the difference and which type is where

# [Window Coordinates]

- Window coordinates start at the left-upper corner of the window
- The method **elem.getBoundingClientRect()** returns window coordinates for elem as an object with properties:
  - **top** – Y-coordinate for the top element edge
  - **bottom** – Y-coordinate for the bottom element edge
  - **left** – X-coordinate for the left element edge
  - **right** – X-coordinate for the right element edge
- When we scroll the page, and the element goes up or down, its *window coordinates* *change*





# [Window Coordinates]

→ Click the button to see its window coordinates:

```
<style>
  body {
    height: 1000px;
  }
  #btnTest {
    margin-top: 200px;
  }
</style>

<button id="btnTest">Show coordinates of this button</button>
<script>
  btnTest.onclick = function () { showRect(this);
  }
  function showRect(elem) {
    let r = elem.getBoundingClientRect(); alert("{top:" + r.top
    + ", left:" + r.left +
    ", right:" + r.right + ", bottom:" + r.bottom + "}");
  }
</script>
```

Show coordinates of this button

→ If you scroll the page, the button position changes, and window coordinates as well

## [elementFromPoint ]

- The call to **document.elementFromPoint(x, y)** returns the most nested element at window coordinates (x, y)
- For instance, the code below highlights and outputs the tag of the element that is now in the middle of the window:

```
let centerX = document.documentElement.clientWidth / 2;  
let centerY = document.documentElement.clientHeight / 2;  
let elem = document.elementFromPoint(centerX, centerY);  
  
elem.style.background = "red";  
  
alert(elem.tagName);
```

- As it uses window coordinates, the element may be different depending on the current scroll position

# [Document Coordinates]

- Document-relative coordinates start from the left-upper corner of the document, not the window
- In CSS, window coordinates correspond to `position:fixed`, while document coordinates are similar to `position:absolute`
- We can use `position:absolute` and `top/left` to put something at a certain place of the document, so that it remains there during a page
- For clarity we'll call window coordinates **(clientX,clientY)** and document coordinates **(pageX,pageY)**

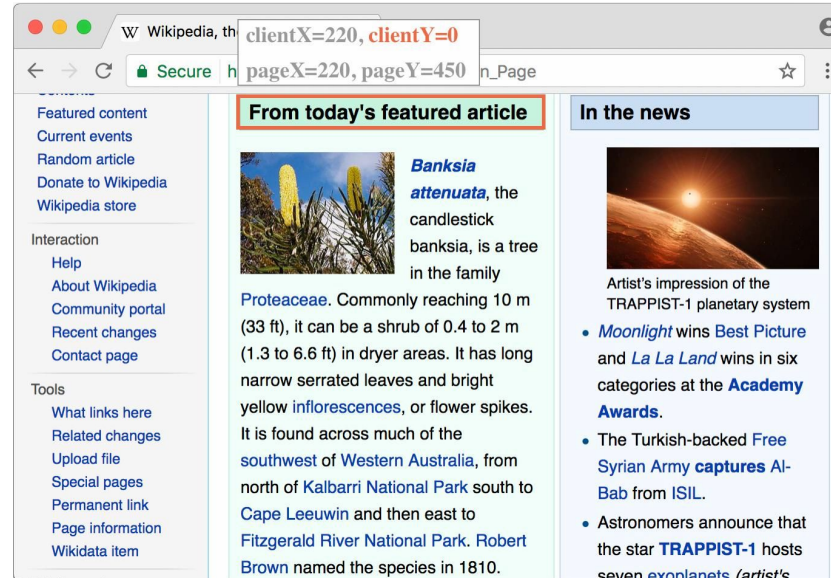
# [Document Coordinates]

- When the page is not scrolled, then window coordinate and document coordinates are actually the same:



# [Document Coordinates]

- And if we scroll the page, then (clientX,clientY) change, because they are relative to the window, but (pageX,pageY) remain the same
- Here's the same page after a vertical scroll:



# [Getting Document Coordinates]

- There's no standard method to get document coordinates of an element
  - But it's easy to write it
- The two coordinate systems are connected by the formula:
  - $\text{pageY} = \text{clientY} + \text{height of the scrolled-out vertical part of the document}$
  - $\text{pageX} = \text{clientX} + \text{width of the scrolled-out horizontal part of the document}$
- The following function takes the window coordinates from `getBoundingClientRect()` and adds the current scroll to them:

```
// get document coordinates of the element function
getCoords(elem) {
  let box = elem.getBoundingClientRect();

  return {
    top: box.top + pageYOffset,
    left: box.left + pageXOffset
  }
}
```

## [ Control questions ]

1. What is Attribute?
2. What is the difference between class and className?
3. How can we move a node?
4. How can we remove a node?
5. What is the difference between clientX and pageX?

# [ Materials ]

Core materials:

<https://learn.javascript.ru/searching-elements-dom>

<https://learn.javascript.ru/searching-elements-internals>

<https://learn.javascript.ru/modifying-document>

Additional materials:

<https://learn.javascript.ru/multi-insert>

<http://learn.javascript.ru/styles-and-classes>

<http://learn.javascript.ru/metrics>

<http://learn.javascript.ru/metrics-window>

<http://learn.javascript.ru/coordinates>

<http://learn.javascript.ru/coordinates-document>

Video materials:

<https://www.youtube.com/watch?v=1FpuKVmotNc>