Document Object Model

Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

What is DOM?

- Document Object Model (DOM) —is a tool by which the JavaScript sees the contents of HTML-page and browser state.

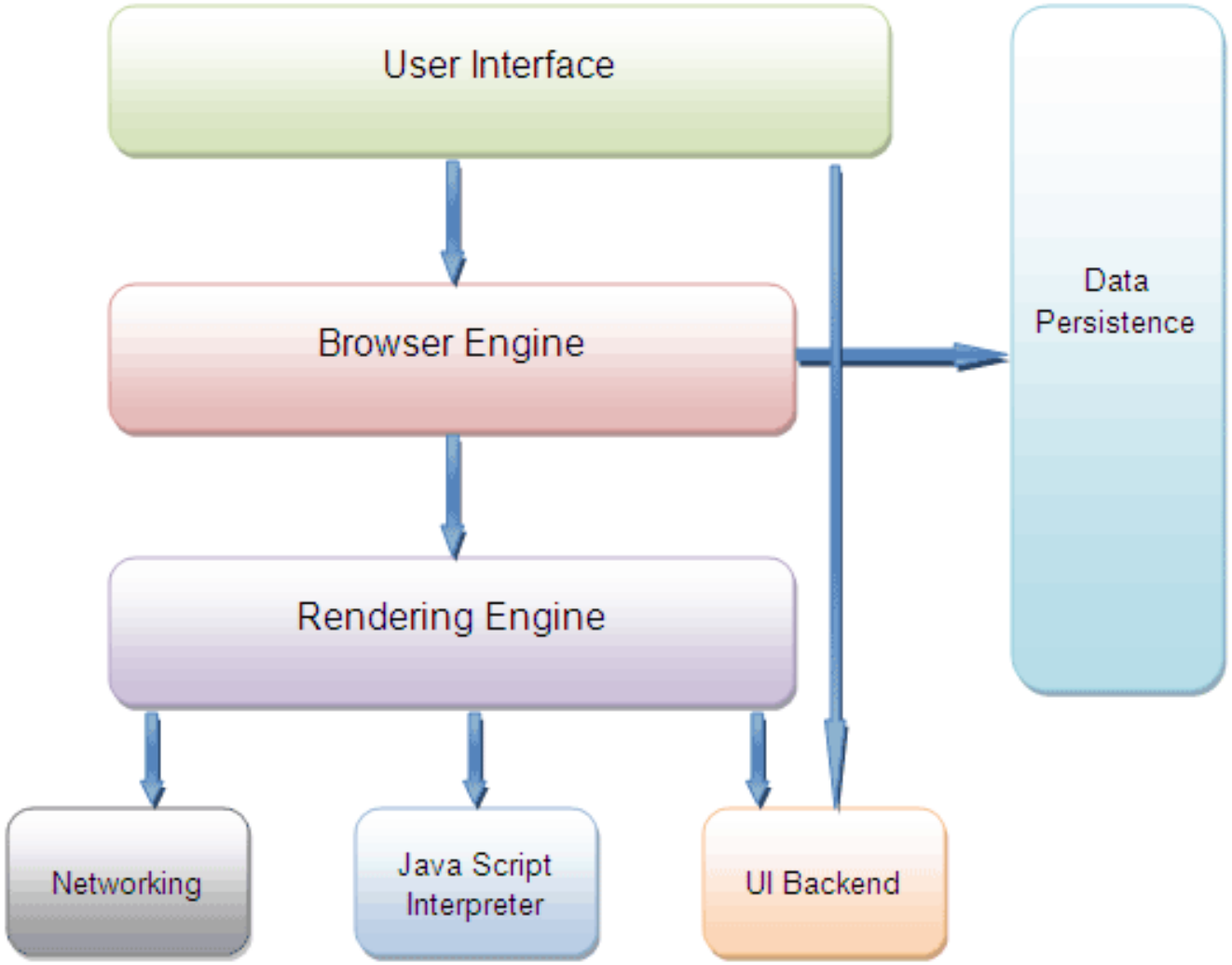
A brief history of DOM

- 1996 - «traditional DOM» or «DOM level 0»
- 1997 - «intermediate DOM»
- 1997-1998 - standard ECMAScript for better
- cross-browser compatibility was released
- 1998 - Internet Explorer 5.0 with limited support DOM level 1 was released
- 2000 - DOM level 2 was published
- 2004 - current version DOM level 2 was published
- 2005 - part W3C DOM was supports by major browsers

W3C DOM Levels

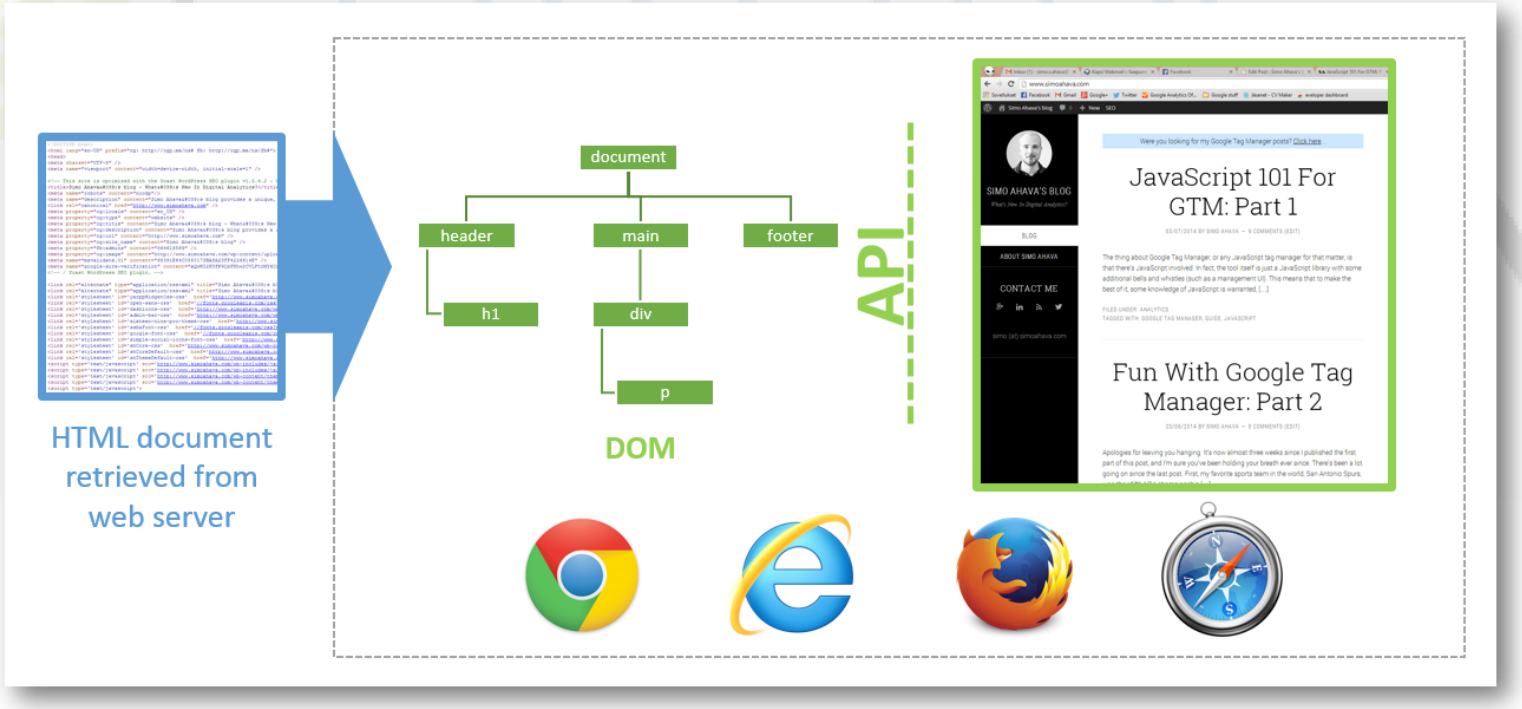
Level Number	Description
Level 0	It includes all the specific models such as: document.images, document.forms, document.layers и document.all. They are not formally specifications DOM, published W3C
Level 1	Basic features DOM (HTML и XML) in documents, such as receiving of node tree of the document, the ability to modify and add data.
Level 2	Support of namespaces XML <--filtered views--> and events
Level 3	It consists of six different specifications that are additional DOM extensions: 1. DOM Level 3 Core; 2. DOM Level 3 Load and Save; 3. DOM Level 3 XPath; 4. DOM Level 3 Views and Formatting; 5. DOM Level 3 Requirements; 6. DOM Level 3 Validation.

A web browser architecture

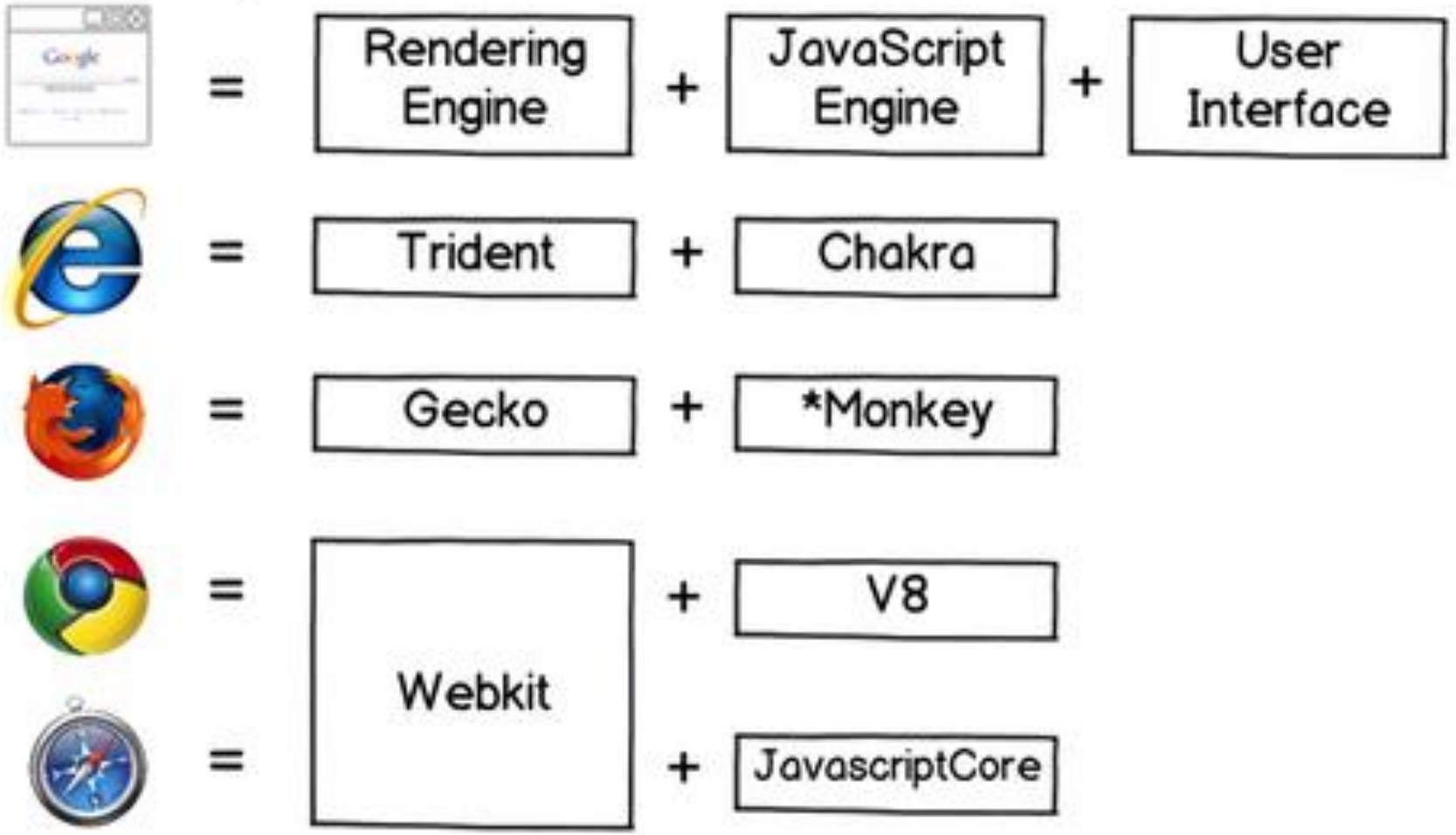


Page rendering flow

Parsing HTML to construct the DOM tree → Render tree construction → Layout of the render tree → Painting the render tree



A web browser engine overview



DOM implementation in web browsers

- checks support the DOM extension in a particular browser (code fragment)

```
function domImplementationTest(){
    var featureArray = ['HTML', 'XML', 'Core', 'Views',
                        'StyleSheets', 'CSS', 'CSS2', 'Events',
                        'UIEvents', 'MouseEvents', 'HTMLEvents',
                        'MutationEvents', 'Range', 'Traversal'];
    var versionArray = ['1.0', '2.0', '3.0'];
    var i;
    var j;
    if(document.implementation && document.implementation.hasFeature){
      for(i=0; i < featureArray.length; i++){
        for(j=0; j < versionArray.length; j++){
          document.write(
            'Поддержка расширения '+ featureArray[i] + ' версии ' + versionArray[j] + ': ' +
            (document.implementation.hasFeature(featureArray[i], versionArray[j]) ?
            '<font style="color:green">true</font>': '<font style="color:red">false</font>') + '<br/>'
          );
        }
        document.write('<br/>');
      }
    }
  }
```

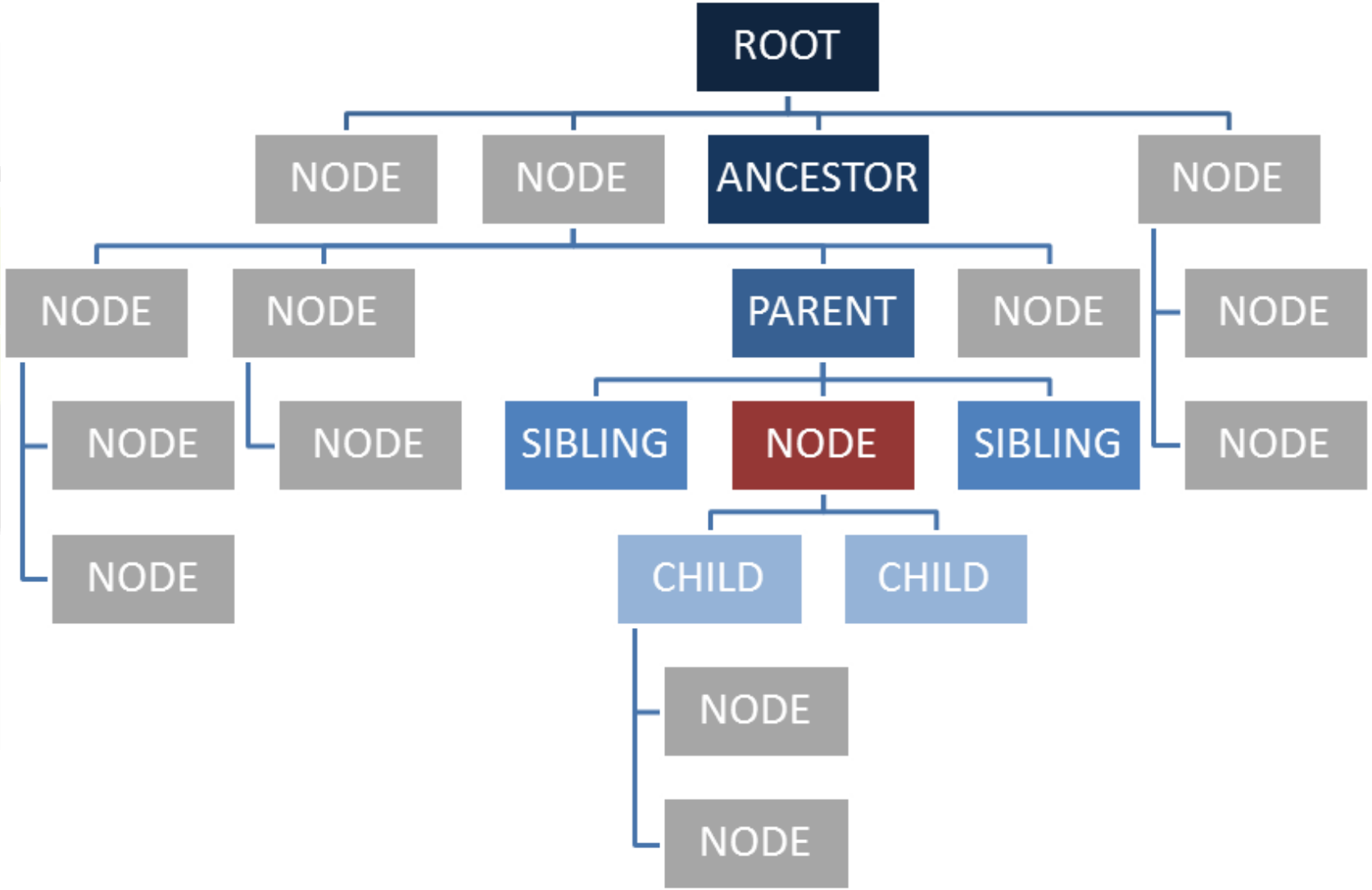
Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

DOM – Terms of Tree Structure

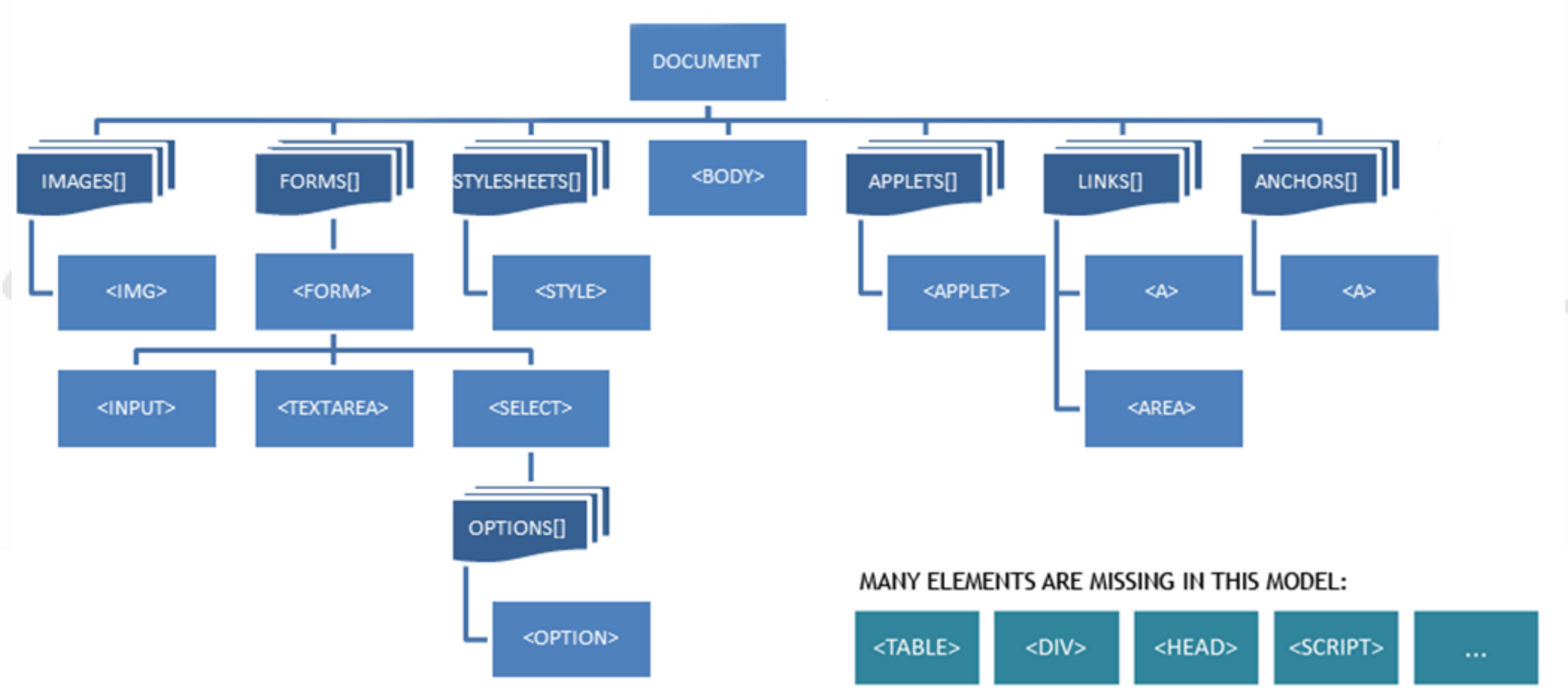


Document Object Model (DOM)

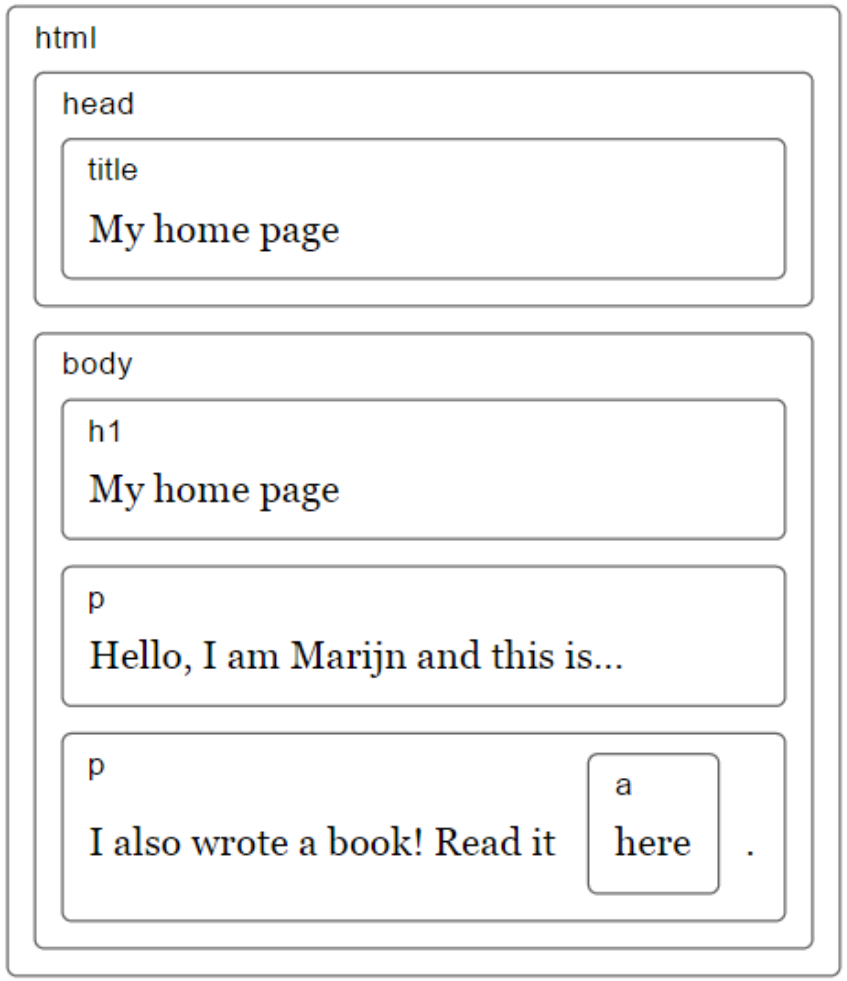
- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

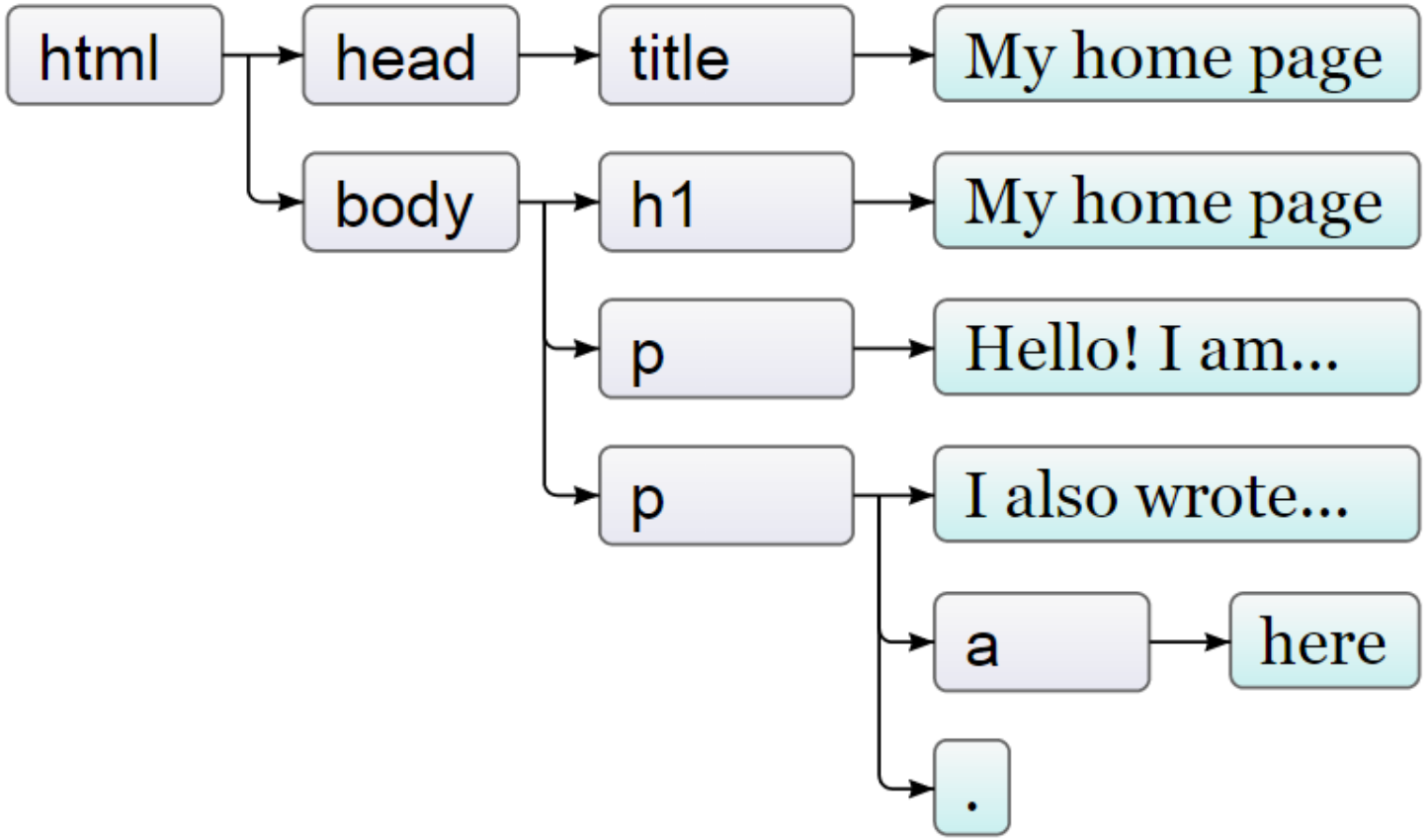
General DOM Tree

- The DOM is one big tree, representing a webpage



DOM Tree Example





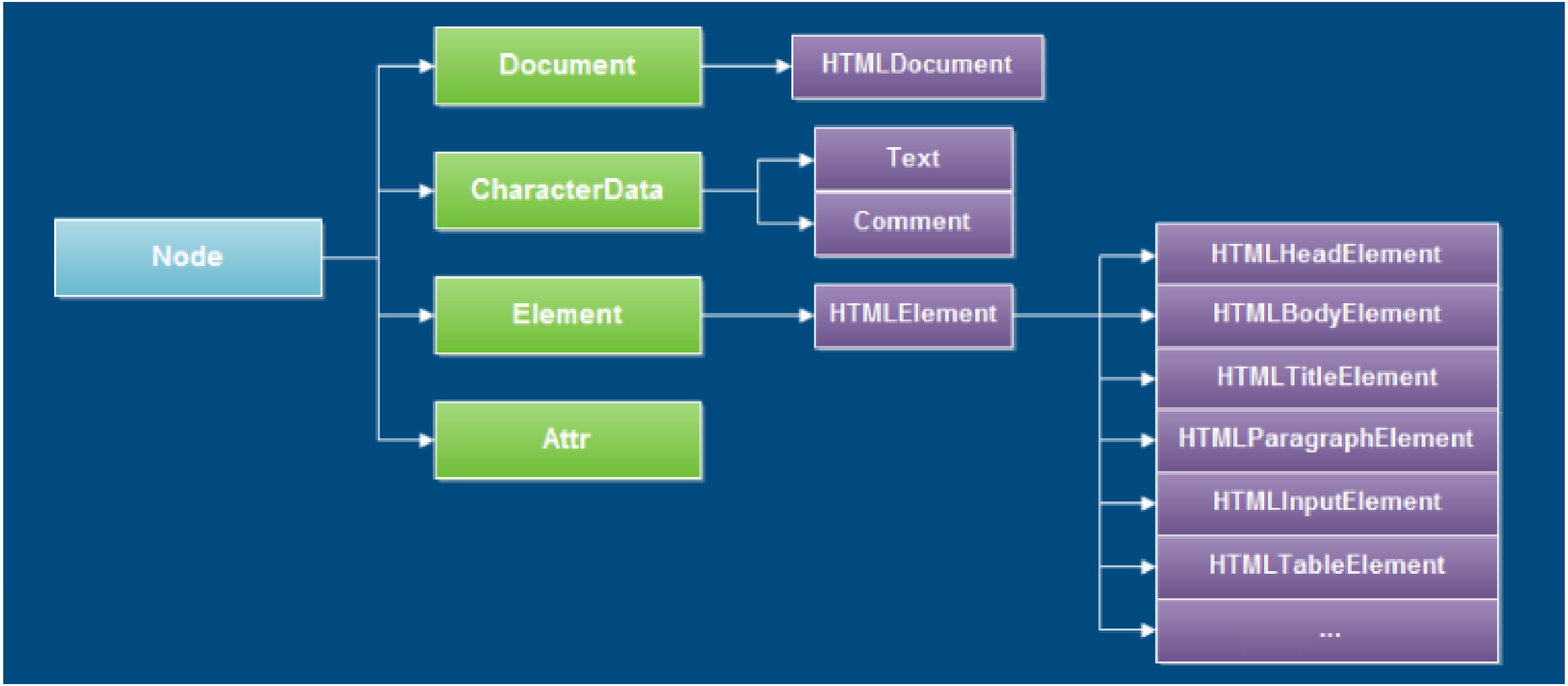
Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

DOM Nodes

- **Nodes** can be an **element** or **collection of elements**.
- Each element of the tree has
  - **properties** like width, value, src etc... (some are read/write, others are read only)
  - functions like submit(), focus() etc.. (which are actually properties too)
  - events like onclick, onchange, onload etc...
- The list is different for each element:
  - the available properties, functions and events depend on **node type** (image, input...)
  - the values of the properties depend on the **state of the node**
  - almost all events are fired because of user interaction

DOM Nodes' Types



DOM Nodes' Types Values

Constant	Value	Description
`Node.ELEMENT_NODE`	1	An `Element` node such as `<p>` or `<div>`.
`Node.TEXT_NODE`	3	The actual `Text` of `Element` or `Attr`.
`Node.PROCESSING_INSTRUCTION_NODE`	7	A `ProcessingInstruction` of an XML document such as `<?xml-stylesheet ... ?>` declaration.
`Node.COMMENT_NODE`	8	A `Comment` node.
`Node.DOCUMENT_NODE`	9	A `Document` node.
`Node.DOCUMENT_TYPE_NODE`	10	A `DocumentType` node e.g. `<!DOCTYPE html>` for HTML5 documents.
`Node.DOCUMENT_FRAGMENT_NODE`	11	A `DocumentFragment` node.

DOM Nodes' Types Values - deprecated

Constant	Value	Description
`Node.ATTRIBUTE_NODE`	2	An `Attribute` of an `Element`. The element attributes are no longer implementing the `Node` interface in DOM4 specification.
`Node.CDATA_SECTION_NODE`	4	A `CDATASection`. Removed in DOM4 specification.
`Node.ENTITY_REFERENCE_NODE`	5	An XML Entity Reference node. Removed in DOM4 specification.
`Node.ENTITY_NODE`	6	An XML `<!ENTITY ...>` node. Removed in DOM4 specification.
`Node.NOTATION_NODE`	12	An XML `<!NOTATION ...>` node. Removed in DOM4 specification.

Elements' properties

`nodeType` – type of a node. The most popular: `"1"` – for Elements and `"3"` – for Text elements. Readonly.

`nodeName/tagName` – tag name with capital letters. `nodeName` has special value for not Elements. Readonly.

`innerHTML` – inside html of element. Could be changed.

`outerHTML` – html of element in including element itself. During writing into `elem.outerHTML` - `elem` will store old element.

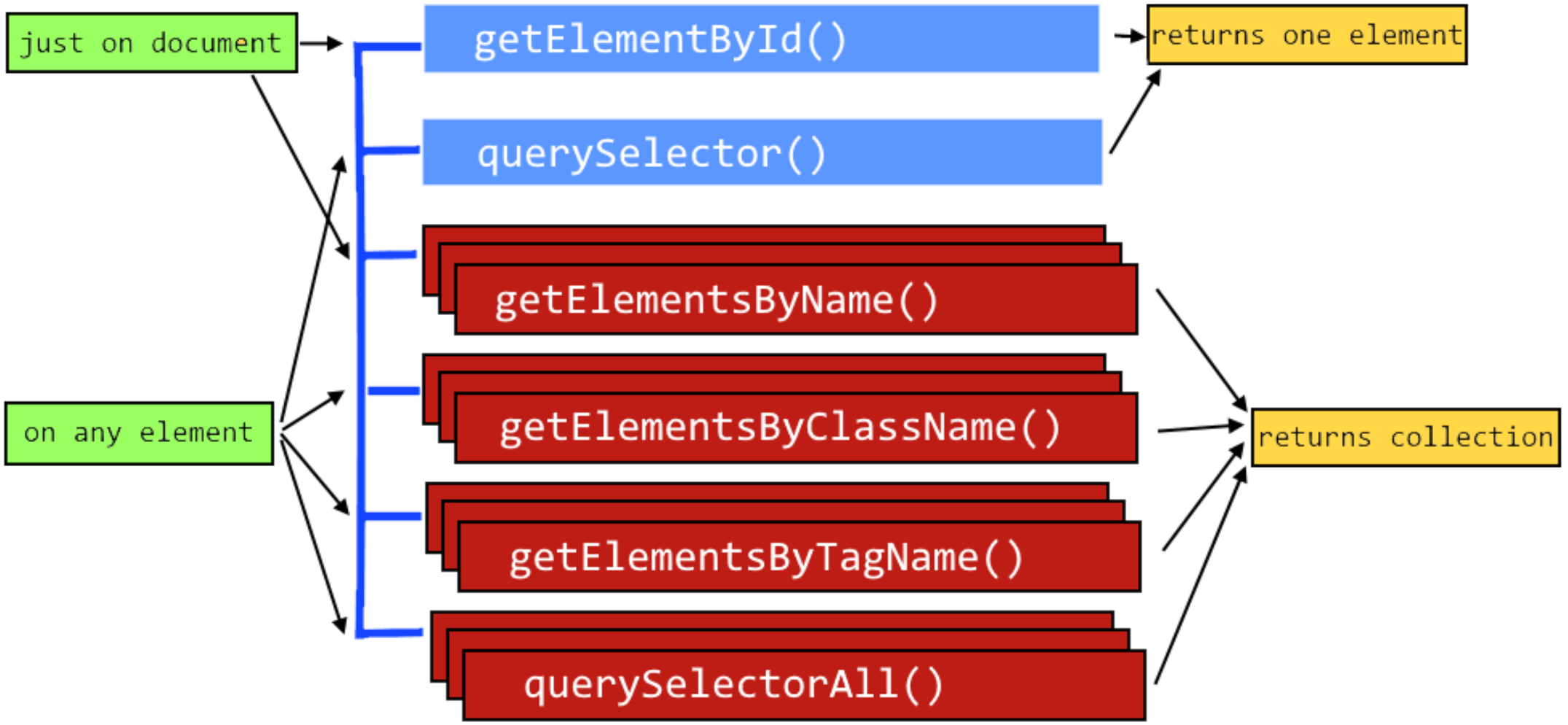
`nodeValue/data` – Value of text element or comment. Property `nodeValue` also is defined for other elements. It could be changed. On some nodes, where `data` is missing, `nodeValue` exists and has value of null, that's why it's better to use `data`.

`textContent` – Text inside element excluding all tags. Could be used to protect bad html injections.

Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

Methods to Perform Search in DOM - 1



Methods to Perform Search in DOM - 2

document.getElementById(elementID)

- get element with id

```
<div id="container">
        <p>Methods to perform search in DOM</p>
        <p>One of such is 'getElementById'</p>
</div>
<script>
        var container = document.getElementById('container');
</script>
```

Methods to Perform Search in DOM - 3

document.getElementsByName(name)

- get all elements with particular name

```
<div name="container">
        <p>Methods to perform search in DOM</p>
        <p name="container">One of such is 'getElementsByName'</p>
</div>
<script>
        var containers = document.getElementsByName('container');
        containers[0].style.color = 'yellow';
</script>
```

Methods to Perform Search in DOM - 4

***element*.getElementsByTagName(name)**

- get all elements with particular tag name

```
<div>
        <p>Methods to perform search in DOM</p>
        <p>One of such is ‘getElementsByTagName’</p>
</div>
<script>
        var paragraphs = document.getElementsByTagName(‘p’);
        for(var i = 0; i < paragraphs.length; i++){
                // paragraphs[i]
        }
</script>
```

Methods to Perform Search in DOM - 5

***element*.getElementsByClassName(className)**

- get all elements with particular class name

```
<div class="container">
        <p>Methods to perform search in DOM</p>
        <p>One of such is 'getElementsByClassName'</p>
</div>
<script>
      var container = document.getElementsByClassName('container')[0];
</script>
```

Methods to Perform Search in DOM - 6

***element*.querySelector(cssSelector)**

- get first element based on a CSS selector

```
<div class="container">
        <p>Methods to perform search in DOM</p>
        <p>One of such is 'queySelector'</p>
</div>
<script>
      var container = document.querySelector('.container');
</script>
```

Methods to Perform Search in DOM - 7

***element*.querySelectorAll(cssSelector)**

- get all elements based on a CSS selector

```
<div  class="container">
        <p>Methods to perform search in DOM</p>
        <p>One of such is 'queySelector'</p>
</div>
<script>
      var paragraphs = document.querySelectorAll('.container p');
</script>
```

Methods to Perform Search in DOM - 7

```
<p>And if you go chasing  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a
        <span class="character">hookah smoking
                <span class="animal">caterpillar</span>
        </span>
</p>
<p>Has given you the call</p>
<script>
    function count(selector) {
        return document.querySelectorAll(selector).length;
    }
    console.log(count("p"));          // All <p> elements  // → 4
    console.log(count(".animal"));    // Class animal   // → 2
    console.log(count("p .animal"));  // Animal inside of <p>   // → 2
    console.log(count("p > .animal")); // Direct child of <p>   // → 1
</script>
```

Methods to Perform Search in DOM - 8

querySelectorAll VS getElementsBy*

```
<ul id='list'>
        <li>item 1</li>
        <li>item 2</li>
</ul>
<script>
        var items1 = document.getElementsByTagName('li');
        var items2 = document.querySelectorAll('li');
        list.innerHtml = ''; //remove all li items
        console.log(items1.length); // 0 – items1 is updated after changes
        console.log(items2.length); // 2 – items2 stayed the same
</script>
```

Summary – Search methods

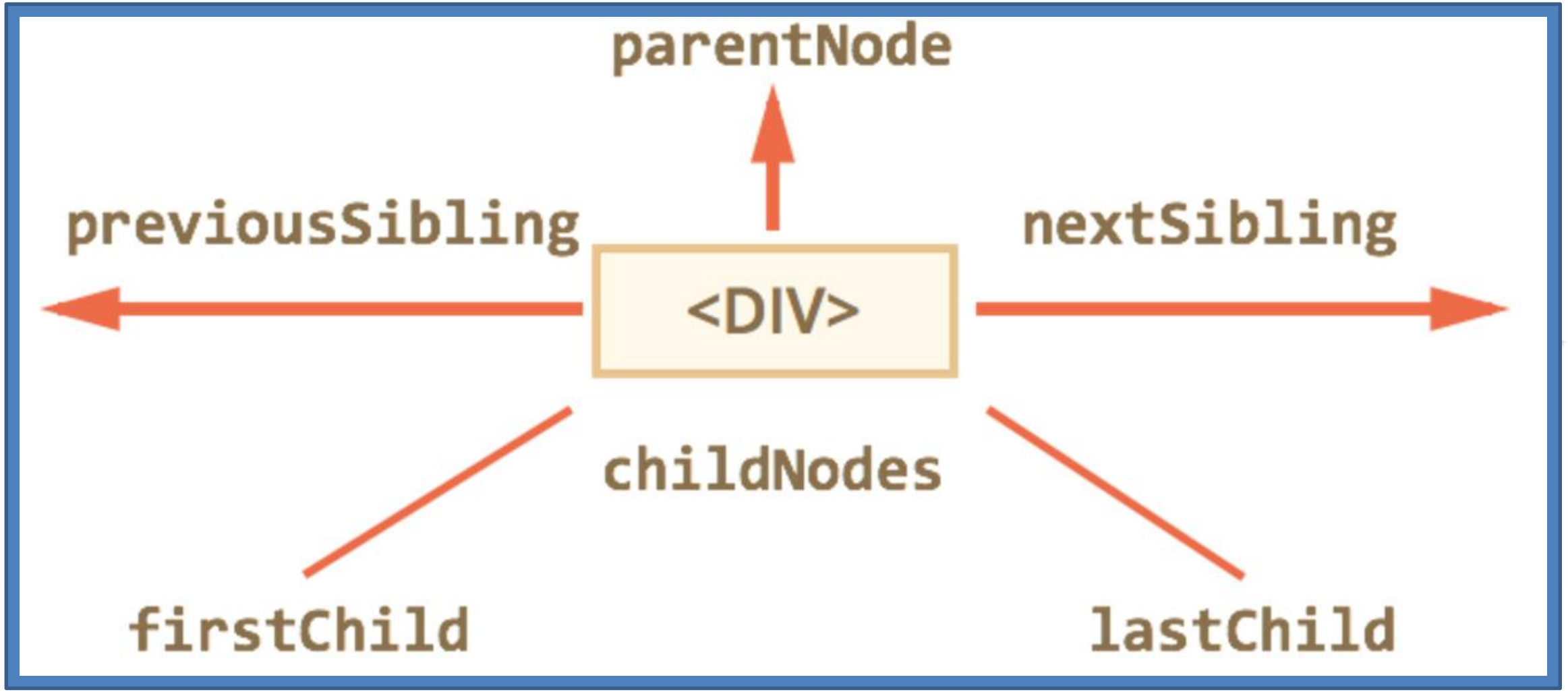
- There are 5 main ways of querying DOM:

  1. `getElementById`
  2. `getElementsByName`
  3. `getElementsByTagName`
  4. `getElementsByClassName` (except IE<9)
  5. `querySelector(All)` (except IE<8 and IE8 in compat mode)

- Some of them can search inside any other element.

- All of them, excepts querySelectorALL, return live collections.

Properties to Navigate in DOM (through all items)



Properties to Navigate in DOM (through all items)

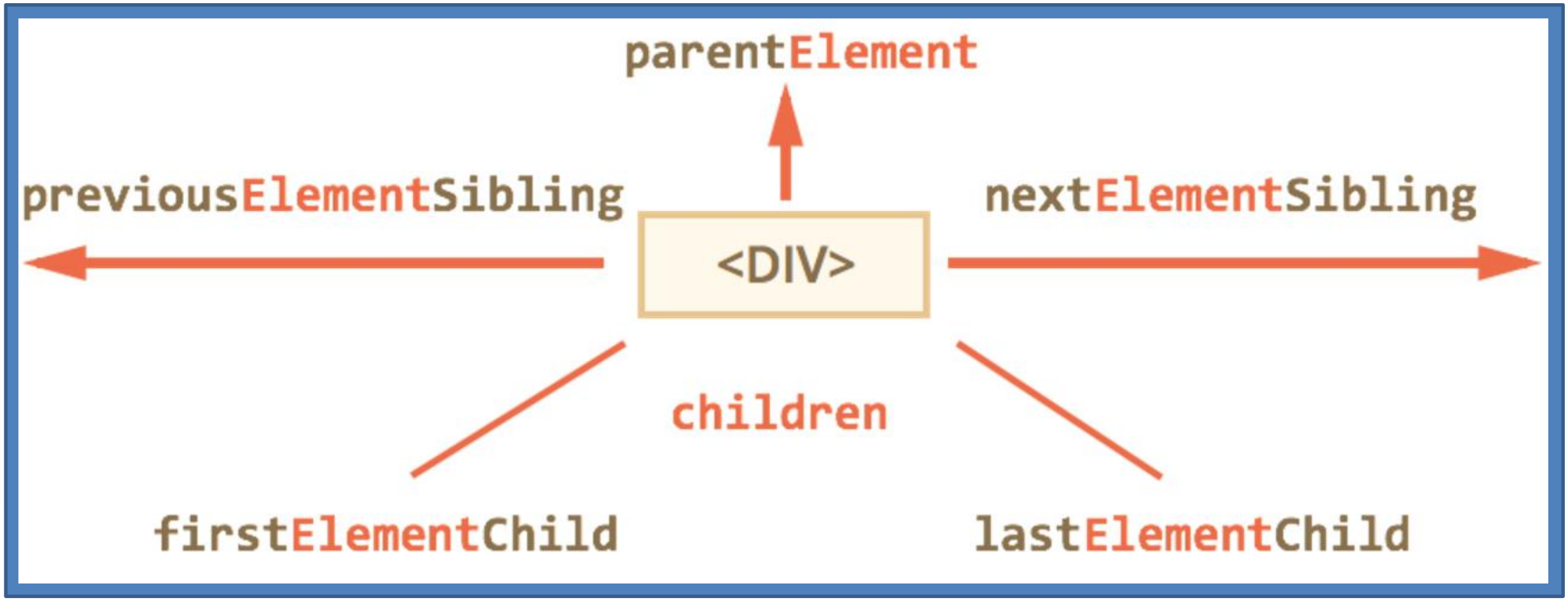
- **Node.childNodes**: You can use this to access all direct child nodes of a single element. It will be an array-like object, which you can loop through. Nodes within this array will include all the different node types including text nodes and other element nodes.

- **Node.firstChild**: This is the same as accessing the first item in the 'childNodes' array ('Element.childNodes[0]'). It's just a shortcut.
- **Node.lastChild**: This is the same as accessing the last item in the 'childNodes' array ('Element.childNodes[Element.childNodes.length-1]'). It's just a shortcut.

- **Node.parentNode**: This gives you access to the parent node of your current node. There will only one parent node. In order to access the grandparent you would simply use 'Node.parentNode.parentNode' etc.

- **Node.nextSibling**: This gives you access to the next node on the same level within the DOM tree.
- **Node.previousSibling**: This gives you access to the last node on the same level within the DOM tree.

Properties to Navigate in DOM (just HTML elements)



Summary – DOM traversing

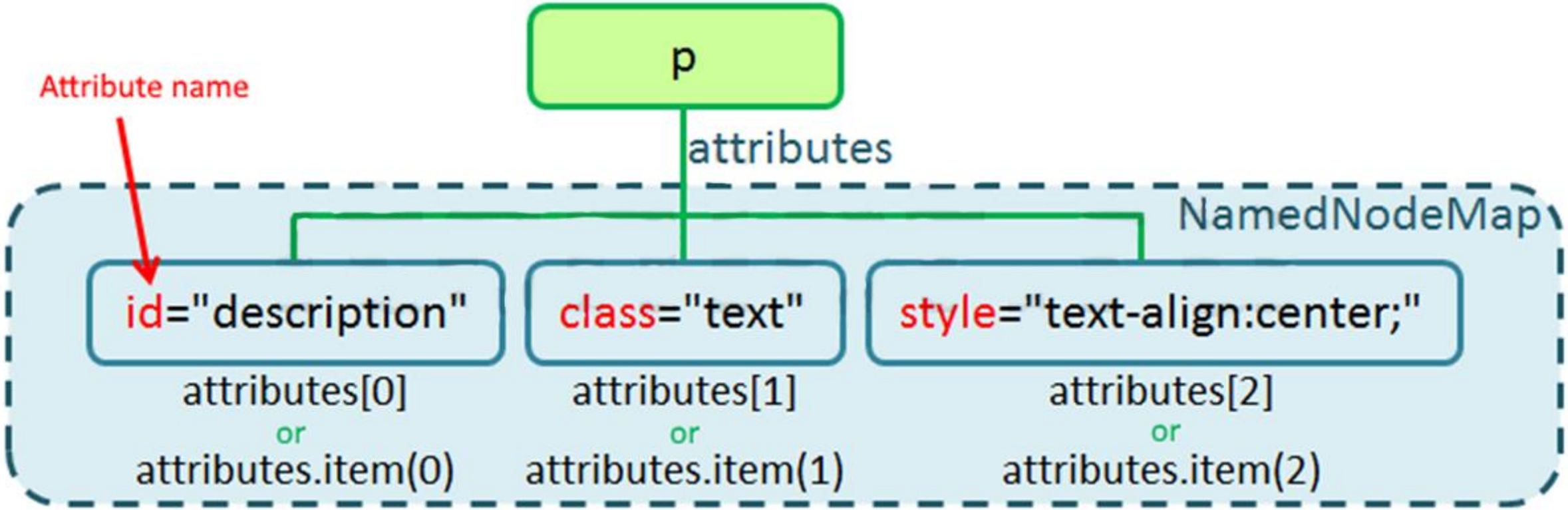
- The DOM tree is tightly interlinked:
  up
    `parentNode/parentElement`
  down
    `children/childNodes`, `firstElementChild/firstChild`,
    `lastElementChild/lastChild`
  left/right
    `previousSibling/nextSibling`,
    `previousElementSibling/nextElemntSibling`

- Browser guarantees that the links are always correct.
- All of them are read-only.
- If there is no such element (child, parent, neighbour etc), the value is `null`.

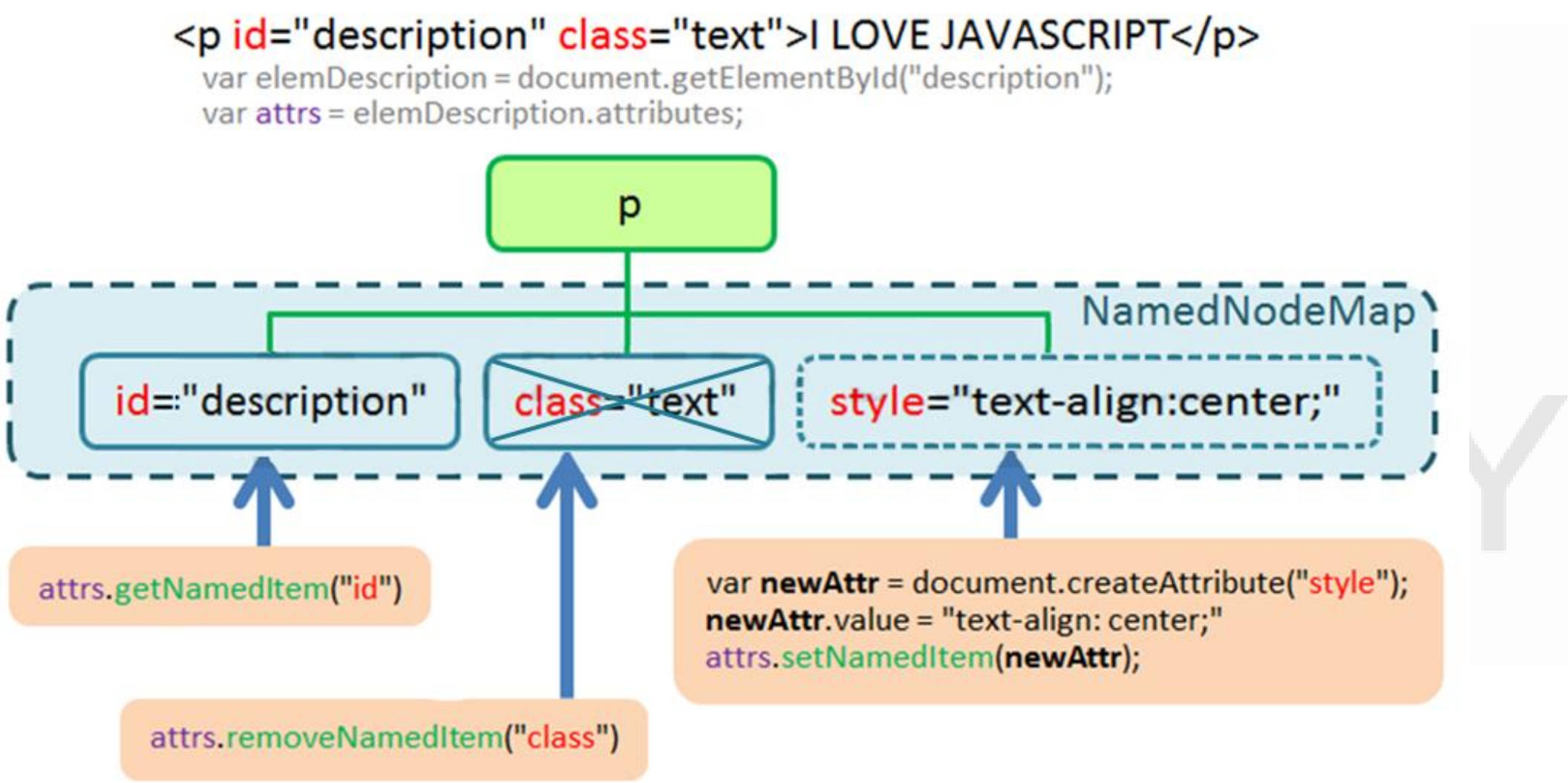
Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

Elements' attributes - 1

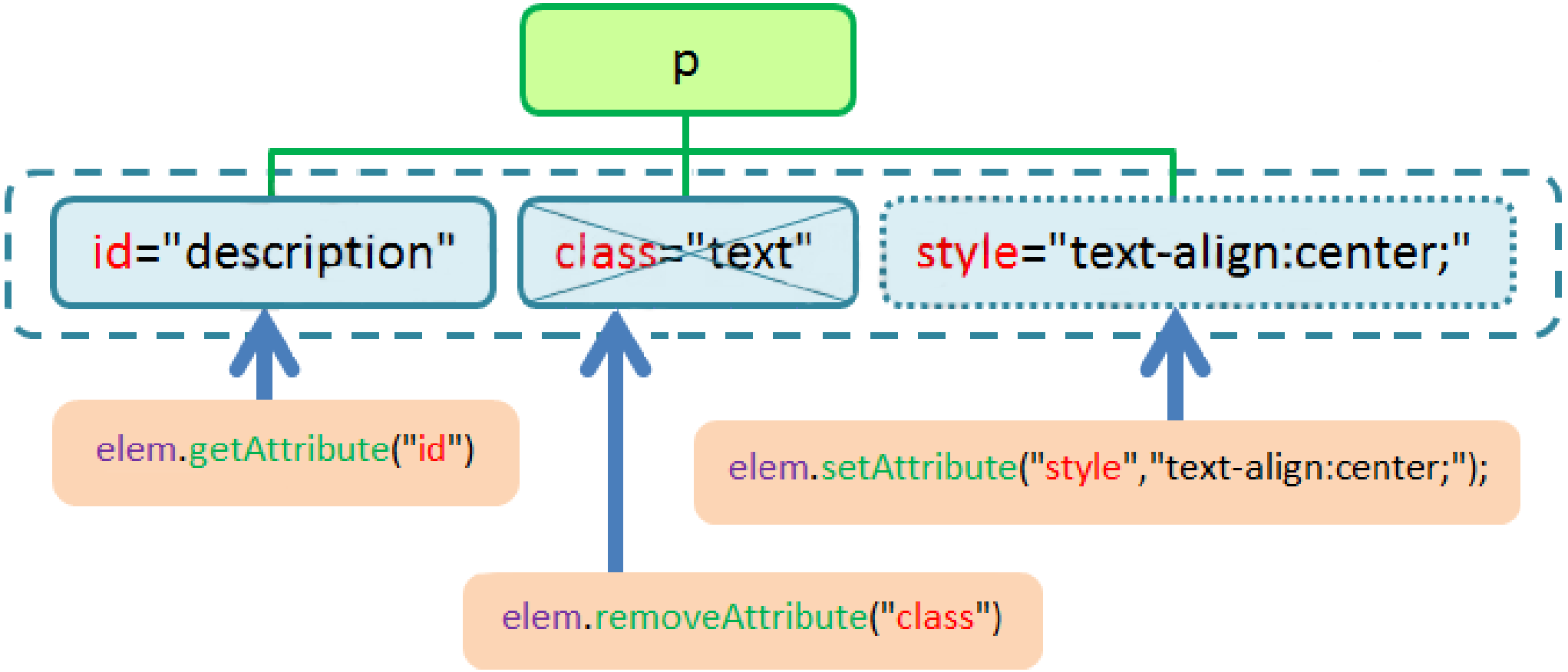


Elements' attributes - 2



Elements' attributes - 3

```
<p id="description" class="text">I LOVE JAVASCRIPT</p>
  var elem = document.getElementById("description");
```



Elements' attributes - 4

In contrast with properties, attributes:

- May be only strings.
- Names not case-sensitive, because HTML attributes are not case-sensitive
- They show up in `innerHTML` (unless it's older IE)
- You can list all attributes using an array-like `attributes` property of the element.

The attributes are broken in IE<8 and in IE8 compatible rendering mode:

- Only `getAttribute` and `setAttribute` methods exist.
- They actually modify DOM properties, not attributes.
- Attributes and properties in IE<8 are merged. Sometimes that leads to weird results.

Attributes as elements' properties

- DOM node is an object. So it can store custom properties and methods just like any JS object.

```
document.body.myData = {    name: 'John‘ };
alert(document.body.myData.name); // John
document.body.sayHi = function() {
        alert(this.nodeName)
};
document.body.sayHi();  // BODY
```

Custom properties and methods are visible only in JavaScript and don't affect HTML. Also, custom properties show up in `for..in` mixed with native properties

- Every type of DOM nodes has standard properties.

id - The element's identifier.
title - The element's advisory title.
lang - Language code defined in RFC 1766.
dir - Specifies the base direction of directionally neutral text and the directionality of tables.
className - The class attribute of the element. This attribute has been renamed due to conflicts with the "class" keyword exposed by many languages.

Example: elements' properties for <a> tag

Each note type could contain specific properties, for example:

<a> tag attributes (or properties):

disabled - Enables/disables the link. This is currently only used for style sheet links, and may be used to activate or deactivate style sheets.
charset - The character encoding of the resource being linked to.
href - The URI of the linked resource.
hreflang - Language code of the linked resource.
media - Designed for use with one or more target media.
rel - Forward link type.
rev - Reverse link type.
target - Frame to render the resource in.
type - Advisory content type.

Attribute Value VS Property Value - 1

- Standard DOM properties are synchronized with attributes.

```
<script>
        document.body.setAttribute('id','la-la-la')
        alert(document.body.id) // la-la-la
</script>
```

- The synchronization does not guarantee the same value.

```
<a href="#"></a>
<script>
        var a  = document.body.children[0];
        a.href = '/‘;
        alert( 'attribute:' + a.getAttribute('href') ) // '/'
        alert( 'property:' + a.href ) // IE: '/', others: full URL
</script>
```

That’s because `href`, according to W3C specification must be a well-formed link.

Attribute Value VS Property Value - 2

- There are other attributes, which are synced, but not copied.

```
<input type="checkbox" checked>
<script>
        var input  = document.body.children[0];
        alert( input.checked ); // true
        alert( input.getAttribute('checked') ); // empty string
</script>
```

The value of input.checked property is either true or false, but the attribute has whatever you put into it.

- There are also built-in properties which are synced one-way only.

```
<input type="text" value="markup">
<script>
    var input = document.body.children[0];
    input.setAttribute('value', 'new');
    alert( input.value ); // 'new', input.value changed
</script>
```

```
<input type="text" value="markup">
<script>
    var input = document.body.children[0];
    input.value = 'new‘;
    alert(input.getAttribute('value')) // 'markup', not changed!
</script>
```

The `"value"` attribute keeps the original value after the property was updated, for example when a visitor typed in something. The original value can be used to check if the `input` is changed, or to reset it.

Attribute Value VS Property Value - 3

To live well with any IE, use attributes correctly. Or, in other words, try using properties all the time, until you *really* need an attribute.

And the only times you *really* need an attribute are:

1. To get a custom HTML attribute, because it is not synced to DOM property.

2. To get an “original value” of the standard HTML attribute, like `<INPUT value="...">`.

Summary – Attributes and properties

- Both **attributes** and **properties** are core features in the DOM model.
- The table of **differences** and **relations**:

Properties	Attributes
Any value	String
Names are case-sensitive	not case-sensitive
Don’t show up in innerHTML	Visible in innerHTML
Standard DOM properties and attributes are synchronized, custom are not.	
Attributes are mixed with properties and screwed up in IE<8, IE8 compat. mode.	

- If you want to have custom attributes in HTML, remember that `data-*` attributes are valid in HTML5.
- In real life, in 98% of cases DOM properties are used.
- You should use attributes in only two cases:
  1. A custom HTML attribute, because it is not synced to DOM property.
  2. To access a built-in HTML attribute, which is not synced from the property, and you are sure you need the attribute. For example, `value` in `INPUT`.

Document Object Model (DOM)

- DOM term, history and levels
- Browser engine and rendering flow
- What is a tree structure
- General DOM tree in browser
- Nodes and node types
- Search and traversing in DOM
- Attributes manipulation
- Manipulating DOM (creating, deleting and replacing elements)

Creating New Elements - 1

DOM modifications is the key to making pages dynamic.

It is possible to **construct new page elements** and **fill them on-the-fly**.

`document.createElement(tag)` - Creates a new DOM element of type **node**

```
var divElement = document.createElement('div')
```

`document.createTextNode(text)` - Creates a new DOM element of type **text**

```
var textElem = document.createTextNode('Hello world')
```

Creating New Elements - 2

An element can also be cloned:

`elem.cloneNode(true)` - Clones an element deeply, with all descendants.

```
var divElement = document.firstElementChild.cloneNode(true);
```

Clones div with all children (ul and li items)

`elem.cloneNode(false)` - Clones an element only, with attributes, but without children.

```
var divElement = = document.firstElementChild.cloneNode(false);
```

Clones div just div. It will be without children

Original div

```
<div>
   <ul>
      <li>Apple</li>
      <li>Banana</li>
      <li>Grape</li>
   </ul>
</div>
```

cloned div (deep clone)

```
<div>
   <ul>
      <li>Apple</li>
      <li>Banana</li>
      <li>Grape</li>
   </ul>
</div>
```

```
<div></div>
```

cloned div (shallow clone)

Appending New Elements – 1

parentElem.appendChild(elem) –
Appends `elem` to the children of `parentElem`.
The new node becomes a last child of the `parentElem`.

```
<div>
    <ul>
        <li>Apple</li>
        <li>Banana</li>
        <li>Grape</li>
    </ul>
</div>

<script>
    var newLi = document.createElement('li');
    newLi.appendChild(document.createTextNode('Pineapple'));
    document.getElementsByTagName('ul')[0].appendChild(newLi);
</script>
```

Result:

```
<div>
   <ul>
      <li>Apple</li>
      <li>Banana</li>
      <li>Grape</li>
      <li>Pineapple</li>
   </ul>
</div>
```

New item was added as last child.

Appending New Elements - 2

parentElem.insertBefore(elem, nextSibling) –
Inserts elem into the children of parentElem before the element nextSibling.

```
<div>
    <ul>
        <li>Apple</li>
        <li>Banana</li>
        <li>Grape</li>
    </ul>
</div>
<script>
    var newLi = document.createElement('li');
    newLi.appendChild(document.createTextNode('Pineapple'));
    var list = document.getElementsByTagName('ul')[0];
    var existingChild = list.children[1];
    list.insertBefore(newLi, existingChild);
</script>
```

Result:

```
<div>
   <ul>
      <li>Apple</li>
      <li>Pineapple</li>
      <li>Banana</li>
      <li>Grape</li>
   </ul>
</div>
```

New item was added on second place. Just before 'Banana'.

elem.insertBefore(newElem, null) // same as elem.appendChild(newElem)

Removing Elements

`parentElem.removeChild(elem)` - Remove the `elem` from the children of `parentElem`.

```
<div>
    <ul>
        <li>Apple</li>
        <li>Banana</li>
        <li>Grape</li>
    </ul>
</div>
<script>
    var list = document.getElementsByTagName('ul')[0];
    var existingChild = list.children[1];
    list.removeChild(existingChild);
</script>
```

Result:

```
<div>
   <ul>
       <li>Apple</li>
       <li>Grape</li>
   </ul>
</div>
```

Replacing Elements

parentElem.replaceChild(elem, currentElem) - Replace the child element of parentElem, referenced by currentElem with the elem.

```
<div>
    <ul>
        <li>Apple</li>
        <li>Banana</li>
        <li>Grape</li>
    </ul>
</div>
<script>
    var newLi = document.createElement('li');
    newLi.appendChild(document.createTextNode('Pineapple'));
    var list = document.getElementsByTagName('ul')[0];
    var existingChild = list.children[1];
    list.replaceChild(newLi, existingChild);
</script>
```

Result:

```
<div>
   <ul>
      <li>Apple</li>
      <li>Pineapple</li>
      <li>Grape</li>
   </ul>
</div>
```

Item was replaced

Summary – DOM manipulation

Creation methods:

- `document.createElement(tag)` - creates a new element node.
- `document.createTextNode(value)` - creates a new text node with given value
- `elem.cloneNode(deep)` - clones the element

Inserting and removing methods are called from parent node. All of them return `elem`:

- `parent.appendChild(elem)`
- `parent.insertBefore(elem, nextSibling)`
- `parent.removeChild(elem)`
- `parent.replaceChild(elem, currentElem)`