CS2003 W07
Object models and functions

Özgür Akgün

Slides adapted from Al Dearle and Saleem Bhatti

Functions in JavaScript

- We have already seen a few examples of functions in JavaScript:

```
function oldStyleAdvert( ) {
        alert( “Buy cheese from cheese.com" );
}
```

- Unlike Java, functions in JavaScript are **first-class values**
- This means that like integers, and strings they may be assigned
- They may be stored in variables, arrays and objects
- They may also be passed as parameters to other functions
- We may also define function literals similarly to integer literals.

Parameter Passing

- JavaScript has two methods of parameter passing.
- All of the base types – strings, numbers and booleans are passed by value
- The object types – object, function and array are all passed by reference
- Note that parameters are un-typed
- If the type of a parameter is important you can test it with typeof operator

Function Literals

- Functions are usually introduced with the keyword **function**
- We can define function with function definitions as we have seen:

```
function square(x) { return x * x; }
```

- We could also write:

- var my_func = function(x) { return x * x; }
- var func2 = function(x) {return x * x * x; }
- my_func = func2; // assignment!

- This may seem strange at first but is a powerful programming paradigm – c.f. functional programming languages like Haskell

all examples in web/node/web3

Function Literals and the Function Constructor

- The Function() constructor permits functions to be dynamically created at runtime
- The Function() constructor creates a new Function each time it is called
- The Function() constructor does not use static scoping - they are always defined as if they are *top level* declarations

Lambda Functions (Arrow fns)

- An arrow function expression is a syntactically compact alternative to a regular function declarations
- Syntax:

  (param1,, … paramN) => { statements }

  (param1, … paramN) => expression

Lambda Functions (Arrow fns)

Example - ordinary first class function

```
filter_fn = function(element) {
   return element.length > 3
}

let my_list = ["duck", "cat", "goat"]

console.log( my_list.filter( filter_fn ) )
```

Same example with lambas

```
console.log( ["duck", "cat", "goat"].filter(el => el.length > 3) )
```

both write

['duck', 'goat']

all examples in web/node/web3

Objects

- JavaScript is not an OO language
- However you can create objects which contain data and (effectively) methods over the data
- You can also program in an OO style as we shall see

Creating an empty Object

- To create an empty object we use the new operator:

```
var my_obj = new Object();
```

- Object is a predefined class
- The location my_obj refers to an object containing no properties (fields) and no methods

Object Literals

- We can also create object literals like this:

  - let oz = { name: “Ozgur Akgun”,
    age: 34,
    married: true,
    occupation: “University Lecturer”
    };

- This object contains 4 properties which we can access in the normal manner using the dot operator

```
alert( oz.name );        // will write out Ozgur Akgun
```

Scope in JavaScript

- JavaScript also has a sophisticated version of dynamic scope
- Using this, we can pass scopes around..
- Consider the object I defined earlier as a scope object,

```
var oz =    {       name: “Ozgur Akgun”,
                    age: 34,
                    married: true,
                    occupation: “University Lecturer”
            };
```

- This creates a scope containing some **bindings** – for example between the name age and the value 34

Changing a scope

- We are used to enriching a scope in programming languages by making declarations

  let car = “Fiat 500”;
- This introduces a new name car into the local scope we can then use it in an expression, function etc.
- We can do the same with JavaScript objects!
- So we can write

  oz.car = “Fiat 500”
- Which adds new property to the object referred to by the name oz
- We can also remove properties using the delete operator:

  delete oz.age; // will remove the age property

Scope Example

- Here is a an example which adds and removes bindings and inspects some objects

```
var oz ={name: “Ozgur Akgun”,
         age: 34,
         married: true,
         occupation: “University Lecturer”
        };

function list_properties( obj ) {
        var names = “”;
        for( var i in obj )
                names += i + “\n”;
        alert( names );
}
```

Scope Example

```
list_properties( oz );
oz.car = "Fiat 500";
list_properties( oz );
delete oz.occupation;
list_properties( oz );
oz.age = "Oz is very old";
alert( oz.age );
oz.age = null;
alert( oz.age );
list_properties( oz );
```

We can do the same starting with object

```
var oz = new Object();     // an object with no properties
// add properties - order is unimportant

oz.married = true,
oz.car = "Fiat 500";
oz.name = “Ozgur Akgun”;
oz.age = 34;
oz.occupation: “University Lecturer”;
```

- This will yield an “oz” that is indistinguishable from the one we made earlier

Object Prototypes

```
function Person(name, age, married, occupation) {
  this.name = name
  this.age = age
  this.married = married
  this.occupation = occupation
}

let oz = new Person(“oz”, 34, true, “University Lecturer”)
```

All JavaScript objects inherit properties and methods from a prototype:

- Date objects inherit from Date.prototype
- Array objects inherit from Array.prototype
- Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

Object Methods

- The **prototype** property permits the addition of new properties
- So could write:

```
Person.prototype.car = “Fiat 500"
```

- You can use this to add methods to an object, so can write:

```
Person.prototype.isMarried = function() {
  return this.married
};
```

Associative Objects - Flanagan 8.6

- We can also use strings to dereference an object

  oz.name

  oz[“name”]

  and

  oz[“n” + “am” + “e”]

… are equivalent

- This is useful for dynamically generating dereference operations - for example from user input
- It is also useful for creating associative lookup tables

Classes

- Modern versions of Javascript also supports classes
- Syntax:

```
class MyClass {

  // class methods

  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  …
}
```

Instances

```
class User {

  constructor(name) {
    // invokes the setter
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short.");
      return;
    }
    this._name = value;
  }
}

let user = new User("John");
alert(user.name); // John
user = new User(""); // Name too short.
```

```
Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
});
```

from:
https://javascript.info/class

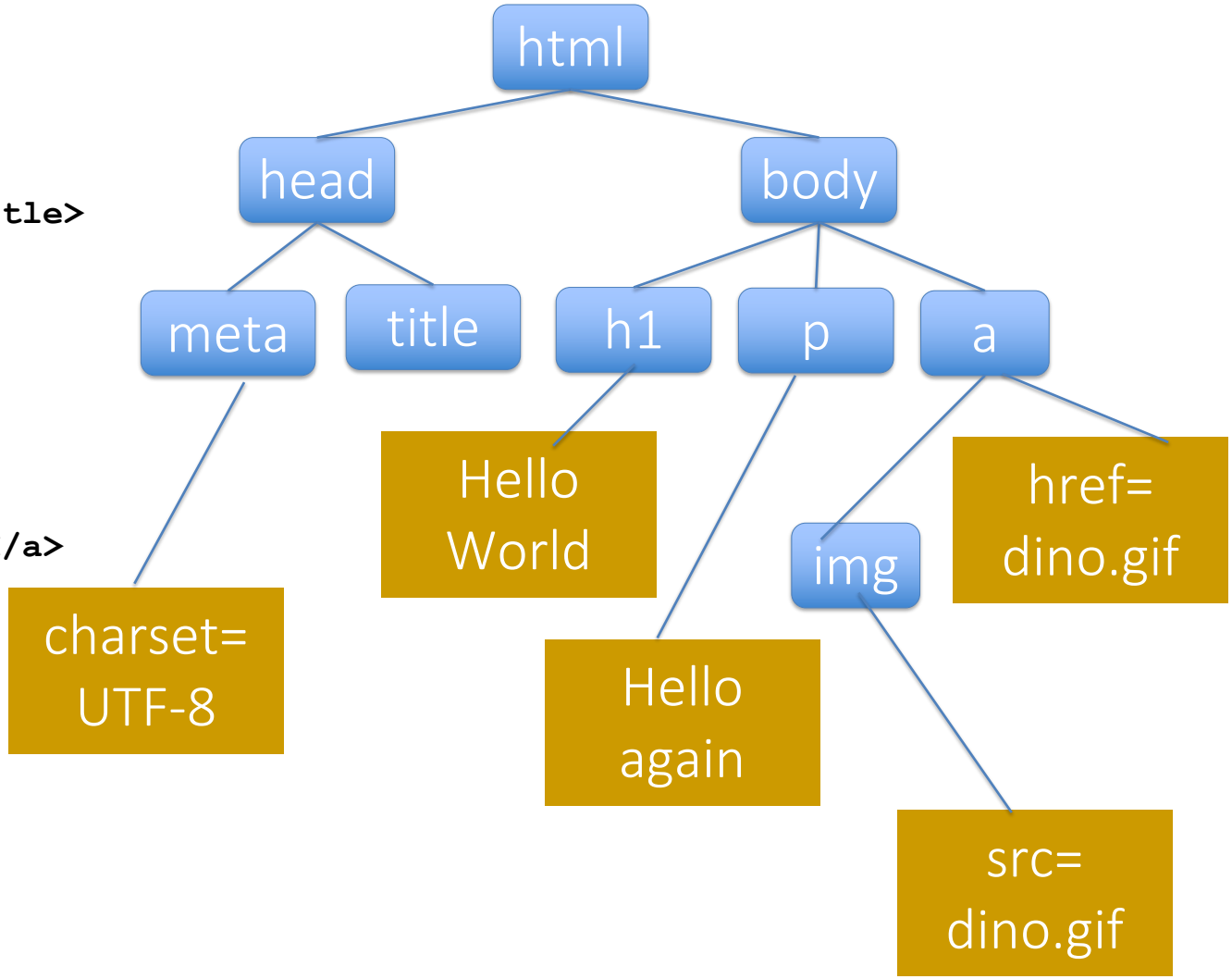
Revisiting the DOM

Document Object Model (DOM)

- Documents have the logical structure of a **tree**.
- Nodes of the tree represent different types of content in a document:
  - as objects with identity, attributes and methods.
- As an object model, the DOM identifies:
  - the interfaces and objects used to represent and manipulate a document.
  - the semantics of these interfaces and objects.
  - the relationships among these interfaces and objects.

DOM– tree / hierarchy of (elements) tags

```
<!DOCTYPE html>
<html>

<head>
<meta charset="UTF-8">
<title>Hello World!</title>
</head>

<body>

<h1>Hello World!</h1>
<p>Hello again</p>
<a href="dino.gif">
<img src="dino.gif" /></a>

</body>

</html>
```



- HTML defines a **data structure** that is accessed and manipulated via the **DOM**.
- The **Javascript DOM API** allows access to the the DOM at the client-side.
- Actions (code) are bound to events (from user actions).

- The Javascript DOM API allows access to all aspects of the DOM:
  - elements
  - attributes
  - values (of elements and attributes)
- **Nodes in the DOM tree can be created, deleted, changed, moved, inserted, removed.**

Key concepts

- **`window`**
  - the OS window hosting the page
- **`navigator`**
  - the browser application in use
- **`screen`**
  - the physical monitor in use
- **`document`**
  - the HTML currently displayed

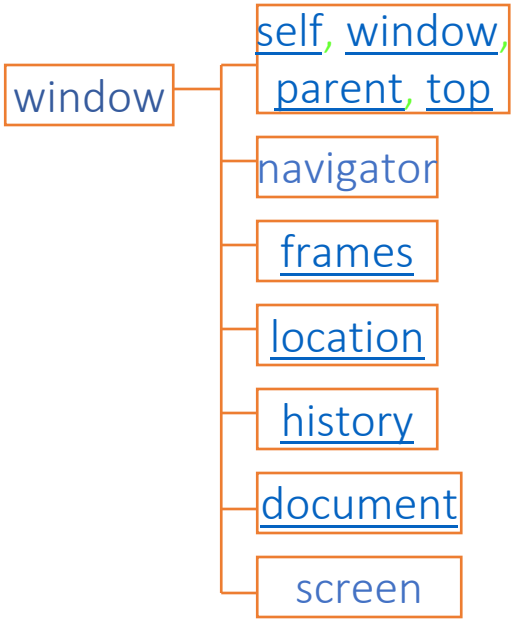
- These are all objects accessible via the **Javascript DOM API.**
- The most commonly used object is:
  - **`document`**
  - in a few slides..
- The other environment information could be used for adjusting presentation or layout

nice clickable interface -> https://www.w3schools.com/jsref/

Window

- In every JS program there is a **global object**
- When JS is executing in a browser, this object is the **Window** in which execution occurs
- This window object contains important objects such as the document object which refers to the HTML contained in the window
- Each window or frame in a JS program has a <u>unique</u> window object

Window



Various window objects

Navigator object

Array of Window objects

Location object - the URL of the current doc
Setting this causes a page load

History object

Document object currently displayed

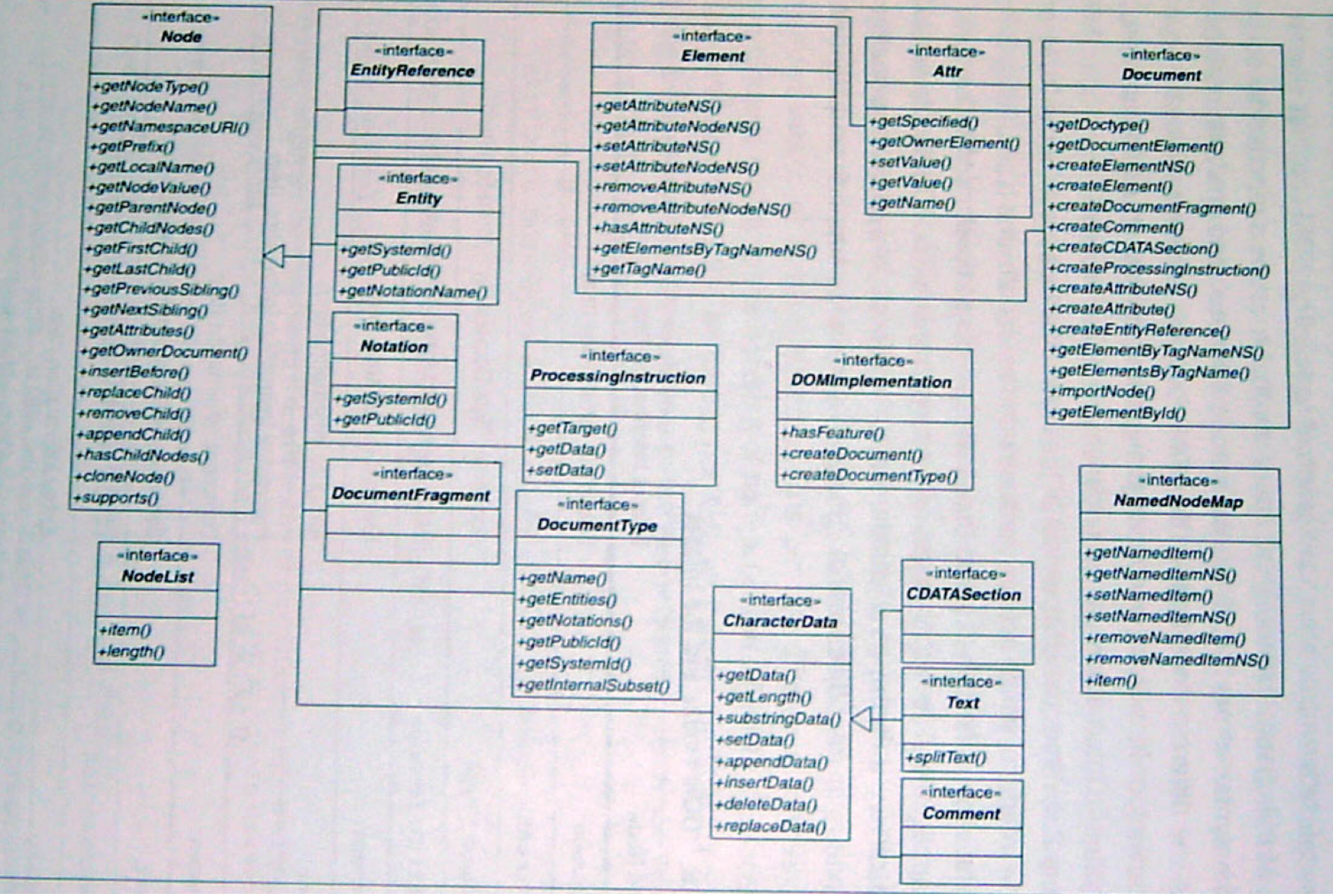
Other properties include.. opener (refers to another window), closed, status (& default status), name, various height properties, amount document has been scrolled..

Navigator

- The Navigator class includes:
  - The name of the browser
  - The platform e.g. MacIntel
  - The (human) language of the browser e.g. Greek
  - If the browser is online
  - The geolocation
  - If cookies are enabled

Screen

- Properties of screen include..
- width
- height
- availWidth
- availHeight
- colourDepth
- pixelDepth

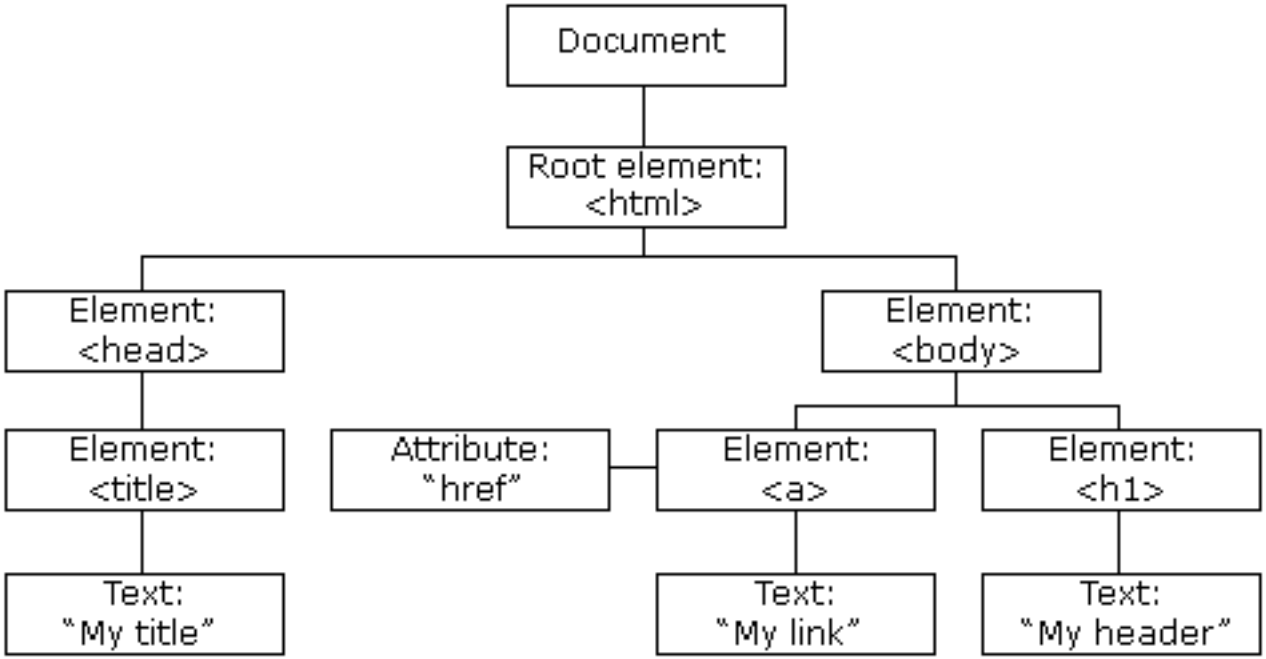

«interface»
Node
+getNodeType()
+getNodeName()
+getNamespaceURI()
+getPrefix()
+getLocalName()
+getNodeValue()
+getParentNode()
+getChildNodes()
+getFirstChild()
+getLastChild()
+getPreviousSibling()
+getNextSibling()
+getAttributes()
+getOwnerDocument()
+insertBefore()
+replaceChild()
+removeChild()
+appendChild()
+hasChildNodes()
+cloneNode()
+supports()
«interface»
EntityReference
«interface»
Element
+getAttributeNS()
+getAttributeNodeNS()
+setAttributeNS()
+setAttributeNodeNS()
+removeAttributeNS()
+removeAttributeNodeNS()
+hasAttributeNS()
+getElementsByTagNameNS()
+getTagName()
«interface»
Attr
+getSpecified()
+getOwnerElement()
+setValue()
+getValue()
+getName()
«interface»
Document
+getDoctype()
+getDocumentElement()
+createElementNS()
+createElement()
+createDocumentFragment()
+createComment()
+createCDATASection()
+createProcessingInstruction()
+createAttributeNS()
+createAttribute()
+createEntityReference()
+getElementByTagNameNS()
+getElementsByTagName()
+importNode()
+getElementById()
«interface»
Entity
+getSystemId()
+getPublicId()
+getNotationName()
«interface»
Notation
+getSystemId()
+getPublicId()
«interface»
ProcessingInstruction
+getTarget()
+getData()
+setData()
«interface»
DOMImplementation
+hasFeature()
+createDocument()
+createDocumentType()
«interface»
DocumentFragment
«interface»
DocumentType
+getName()
+getEntities()
+getNotations()
+getPublicId()
+getSystemId()
+getInternalSubset()
«interface»
NamedNodeMap
+getNamedItem()
+getNamedItemNS()
+setNamedItem()
+setNamedItemNS()
+removeNamedItem()
+removeNamedItemNS()
+item()
«interface»
NodeList
+item()
+length()
«interface»
CharacterData
+getData()
+getLength()
+substringData()
+setData()
+appendData()
+insertData()
+deleteData()
+replaceData()
«interface»
CDATASection
«interface»
Text
+splitText()
«interface»
Comment

Node

- Most of the objects in DOM inherit from the Node class
- It has many properties (and methods to manipulate them).
  - attributes A NamedNodeMap containing the attributes of this node (if it is an Element)
  - baseURI Returns the absolute base URI of a node
  - nodeType Returns the type of a node
  - nodeValue Sets or returns the value of a node, depending on its type
  - ownerDocument Returns the root element (document object) for a node
  - nodeName Returns the name of a node, depending on its type
  - textContent Sets or returns the textual content of a node and its descendants
  - prefix Sets or returns the namespace prefix of a node

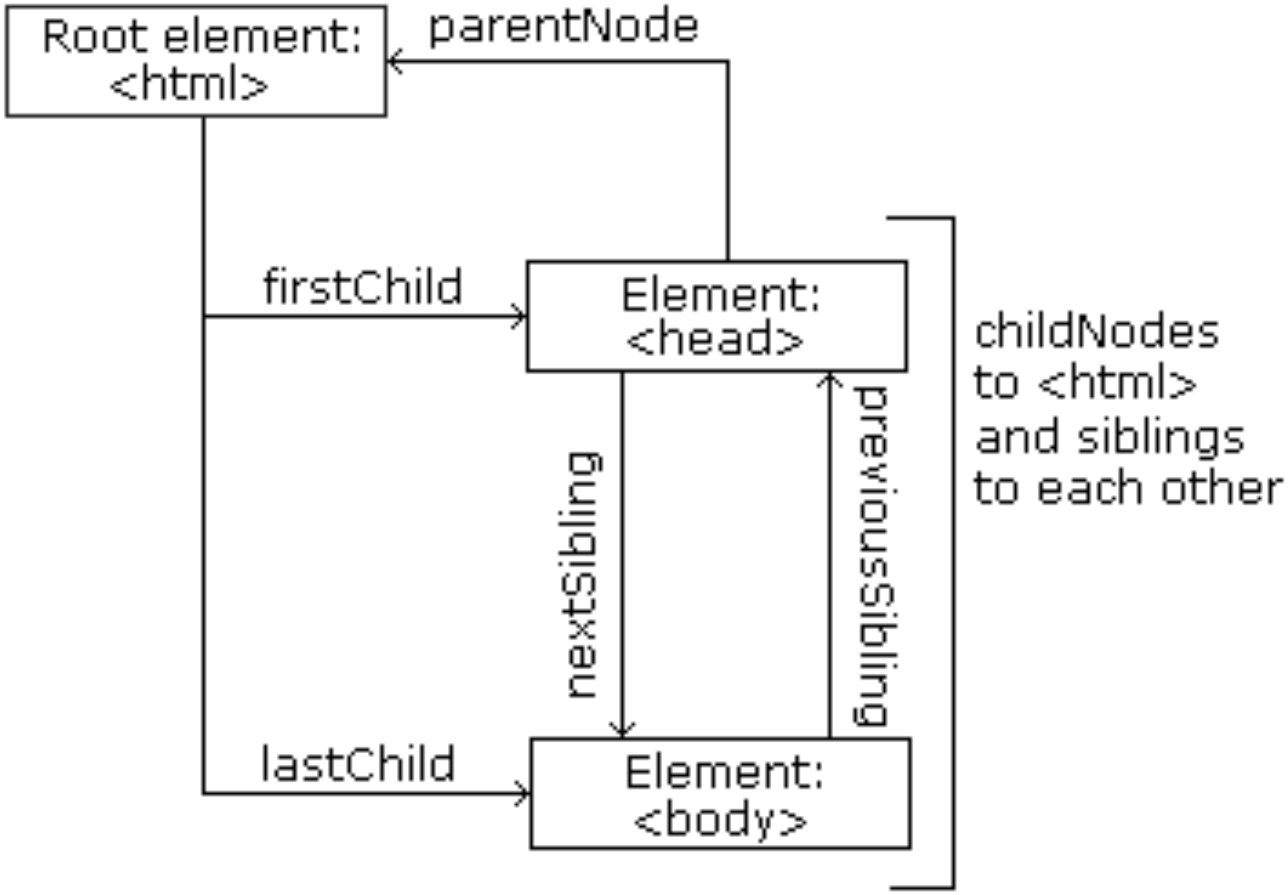
  - childNodes Returns a NodeList of child nodes for a node
  - firstChild Returns the first child of a node
  - lastChild Returns the last child of a node
  - nextSibling Returns the node immediately following a node
  - parentNode Returns the parent node of a node
  - previousSibling Returns the node immediately before a node

Parent-child relationship



from https://www.w3schools.com/js/js_htmldom_navigation.asp

Parent-child relationship



https://www.w3schools.com/js/js_htmldom_navigation.asp

Parent-child relationship

```
<html>

  <head>
    <title>DOM</title>
  </head>

  <body>
    <h1>DOM lesson</h1>
    <p>Hello world!</p>
  </body>

</html>
```

From the HTML above you can read:

- <html> is the root node
- <html> has no parents
- <html> is the parent of <head> and <body>
- <head> is the first child of <html>
- <body> is the last child of <html>

and:

- <head> has one child: <title>
- <title> has one child (a text node): "DOM Tutorial"
- <body> has two children: <h1> and <p>
- <h1> has one child: "DOM Lesson one"
- <p> has one child: "Hello world!"
- <h1> and <p> are siblings

but usually just use getElementById() etc.

https://www.w3schools.com/js/js_htmldom_navigation.asp

Document

- The **`document`** object provides the starting point to access the HTML tags on a page.
- It is the root of the tree:
  - parent to the <html> tag.
- The `document` object is always accessible directly via Javascript.
- The API for `document` provides access to an individual `Node`.

DOM Document in JavaScript

Document has the all the properties and methods of the Node object as well as the properties and methods defined below.

The Document object has the following properties:

- doctype
- implementation
- documentElement

DOM Document in JavaScript

The Document object has the following methods:

- createElement(tagName)
- createDocumentFragment()
- createTextNode(data)
- createComment(data)
- createCDATASection(data)
- createProcessingInstruction(target, data)
- createAttribute(name)
- createEntityReference(name)
- getElementsByTagName(tagname)
- importNode(importedNode, deep)
- createElementNS(namespaceURI, qualifiedName)
- createAttributeNS(namespaceURI, qualifiedName)
- getElementsByTagNameNS(namespaceURI, localName)
- getElementById(elementId)