Teamnet transforming technology

JavaScript fundamentals

Pregatit de: Daniel Vișoiu

Data: 04.05.2015

Agenda

1. Introduction
2. JavaScript basics
3. Data structures and types
4. Functions
5. Built-in types
6. Control flow and error handling
7. DOM interactions

1

Introduction

What is JavaScript?

- A cross-platform, object-oriented scripting language
- Small and lightweight
- The language for web pages
- Is **NOT** Java (but they do have some similarities)

History

- created in 1995 by Brendan Eich, an engineer at Netscape, as a way to add programs to web pages
- introduced in 1996 with the second version of Netscape Navigator browser
- Netscape submitted the language to Ecma International (European Computer Manufacturers Association), which resulted in the **ECMAScript** standard in 1997
- is one of the major implementations of ECMAScript (other implementations are **ActionScript** (Adobe), **JScript** (Microsoft))
- current stable release: 1.8.5 (March 2011)

Why use JavaScript?

It adds behaviour to the web page making it capable of responding to actions without needing to load a new web page:

- form validation
- loading new images, objects or scripts
- improving user experience

2

Basics

Basics

- **JavaScript** borrows most of its syntax from Java, but is also influenced by Awk, Perl and Python.
- Is **case-sensitive** and uses the Unicode character set
- Spaces, tabs and newline characters are called whitespace
- Instructions are called **statements** and are separated by a semicolon (;)
- Has rules for automatic insertion of semicolons (ASI) to end statements; but it is recommended to always add **semicolons** to end your statements (it will avoid side effects)

How to use JavaScript

1. **Internal** – using the <script></script> tag

   ```
   <script type="text/javascript">
       alert("Hello world");
   </script>
   ```

2. **Inline**

   ```
   <button onclick="alert('Hello world');"></button>
   ```

3. **External file** – using the <script></script> tag

   ```
   <script src="main.js"></script>
   ```

The web console

- The Web Console shows you information about the currently loaded Web page
- includes a command line that you can use to execute JavaScript expressions in the current page.
- Usually opens with F12

Hello World

```
function hello(user) {
    return "Hello " + user;
}

hello("world"); // "Hello world“
```

Lazy version: `console.log("Hello world");`

Good to know before we start

- ***alert(message)*** – function that shows the given message in a small popup window (with an OK button)

```
      alert('Hello world');
```

- ***console.log(message)*** – function that shows the given message in the **JavaScript console window** (that opens with F12 in most browsers)

```
      console.log("Hello world");
```

- All the proposed exercises will be resolved using external .js files ☺
- There will be no JavaScript code written in the HTML ☺

Comments

```
// single line comment


/*
multiline
comment
*/


alert("Hello World"); //comments can be appended to the end of lines
```

3

Data structures and types

Variables

The names of variables, called **identifiers**, conform to certain rules:

- must start with a letter, underscore (_), or dollar sign ($)
- subsequent characters can also be digits (0-9)
- **case-sensitive**

Some examples of legal names are Number_hits, temp99, and _name

Declarations

There are three kinds of declarations in JavaScript:

1. **var**
   - Declares a variable, optionally initializing it to a value.
2. **let** (not fully supported)
   - Declares a block scope local variable, optionally initializing it to a value.
3. **const** (not fully supported)
   - Declares a read-only named constant.

Data types

The latest **ECMAScript** standard defines seven data types:

- Six data types that are primitives:
  - **Boolean**: true and false
  - **null**: a special keyword denoting a null value. Because JavaScript is case-sensitive, null is not the same as Null, NULL, or any other variant
  - **undefined**: a top-level property whose value is undefined.
  - **Number**: 42 or 3.14159
  - **String**: "Howdy"
  - **Symbol** (new in ECMAScript 6)
- **Object**

Data types

- The **primitives** enable you to perform useful functions with your applications
- **Objects** and **functions** are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your application can perform.

Data type conversion

- JavaScript is a **dynamically typed** language:
  - you don't have to specify the data type of a variable when you declare it
  - data types are converted automatically as needed during script execution.

var answer = 42; //defining a number variable

answer = "Thanks for all the fish..."; //reassingning the variable with a string value

Data type conversion

- In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings.

```
x = "The answer is " + 42 // "The answer is 42"
y = 42 + " is the answer" // "42 is the answer"
```

- In statements involving other operators, JavaScript does not convert numeric values to strings.

```
"37" - 7 // 30
"37" + 7 // "377"
```

Converting strings to numbers

- In the case that a value representing a number is in memory as a string, there are methods for conversion.

- **parseInt**(string, radix)

radix = An integer between 2 and 36 that represents the base in mathematical numeral systems

- **parseFloat**(string)

Variable scope

- **Scope** is the set of variables you have access to.
- There are **two** kinds of scopes

- **Local scope**
  - Variables declared within a JavaScript function, become LOCAL to the function.
  - Local variables have local scope: They can only be accessed within the function.
  - Local variables are created when a function starts, and deleted when the function is completed and they are no longer references.

Variable scope

- **Global scope**
  - A variable declared outside a function, becomes GLOBAL.
  - A global variable has global scope: All scripts and functions on a web page can access it.

- *Automatically Global*

  If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable.

Variable scope

```
// code here can not use carName

function myFunction() {
    var carName = "Mercedes";

    // code here can use carName

}
```

Variable scope

```
var carName = "Mercedes";
// code here can use carName
function myFunction() {
    // code here can use carName
}
```

Variable scope

```
// code here can use carName

function myFunction() {
    carName = "Mercedes";

    // code here can use carName

}
```

Variable hoisting

- Another unusual thing about variables in JavaScript is that you can refer to a variable declared later, without getting an exception.
- This concept is known as **hoisting**; variables in JavaScript are in a sense "hoisted" or lifted to the top of the function or statement.
- However, variables that aren't initialized yet will return a value of undefined.

Variable hoisting

```
console.log(declaredLater);
// Outputs: undefined

var declaredLater = "Now it's defined!";

console.log(declaredLater);
// Outputs: "Now it's defined!"
```

Variable hoisting

```
console.log(getValue());
// Outputs: Hello world!

function getValue() {
    return "Hello world!";
}

console.log(getValue());
// Outputs: Hello world!
```

Literals

- You use **literals** to represent values in JavaScript. These are fixed values, not variables, that you literally provide in your script.

- Literal integers:
  - decimal (base 10) - sequence of digits without a leading 0: 117 and -345
  - octal (base 8) - Leading 0 (zero) on an integer literal indicates it is in octal: 015, 0001 and -077
  - hexadecimal (base 16) - Leading 0x (or 0X) indicates hexadecimal: 0x1123, 0x00111 and -0xF1A7

String literals

- A string literal is zero or more characters enclosed in double (") or single (') quotation marks.
- A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or both double quotation marks. The following are examples of string literals:

```
"foo"
'bar'
"1234"
"one line \n another line"
"John's cat"
```

Object literals

- An **object** literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({})

```
var sales = “Toyota”;
var car = { myCar: "Saturn", cost: 15000, special: sales };

console.log(car.myCar);   // Saturn
console.log(car.cost);  // 15000
console.log(car.special); // Toyota
```

Equality

- Objects are only equal to themself
- Primitives are equal if the values match (“cat” === “cat”)
- Two sets of equality operators (== and ===)
  - == performs **type coercion** if you give it different types
    - "dog" == "dog"; // true
    - 1 == true; // true
  - === avoids **type coercion**
    - 1 === true; // false
    - true === true; // true

Truthy and Falsy values

- The following values will evaluate to **false** (are falsy):
  - false
  - undefined
  - null
  - 0
  - NaN
  - the empty string ("")

- All other values, including all objects evaluate to **true** (are truthy)

- To test the Truthy/Falsy value of an ***val*** variable simply use double negation: console.log(!!val);

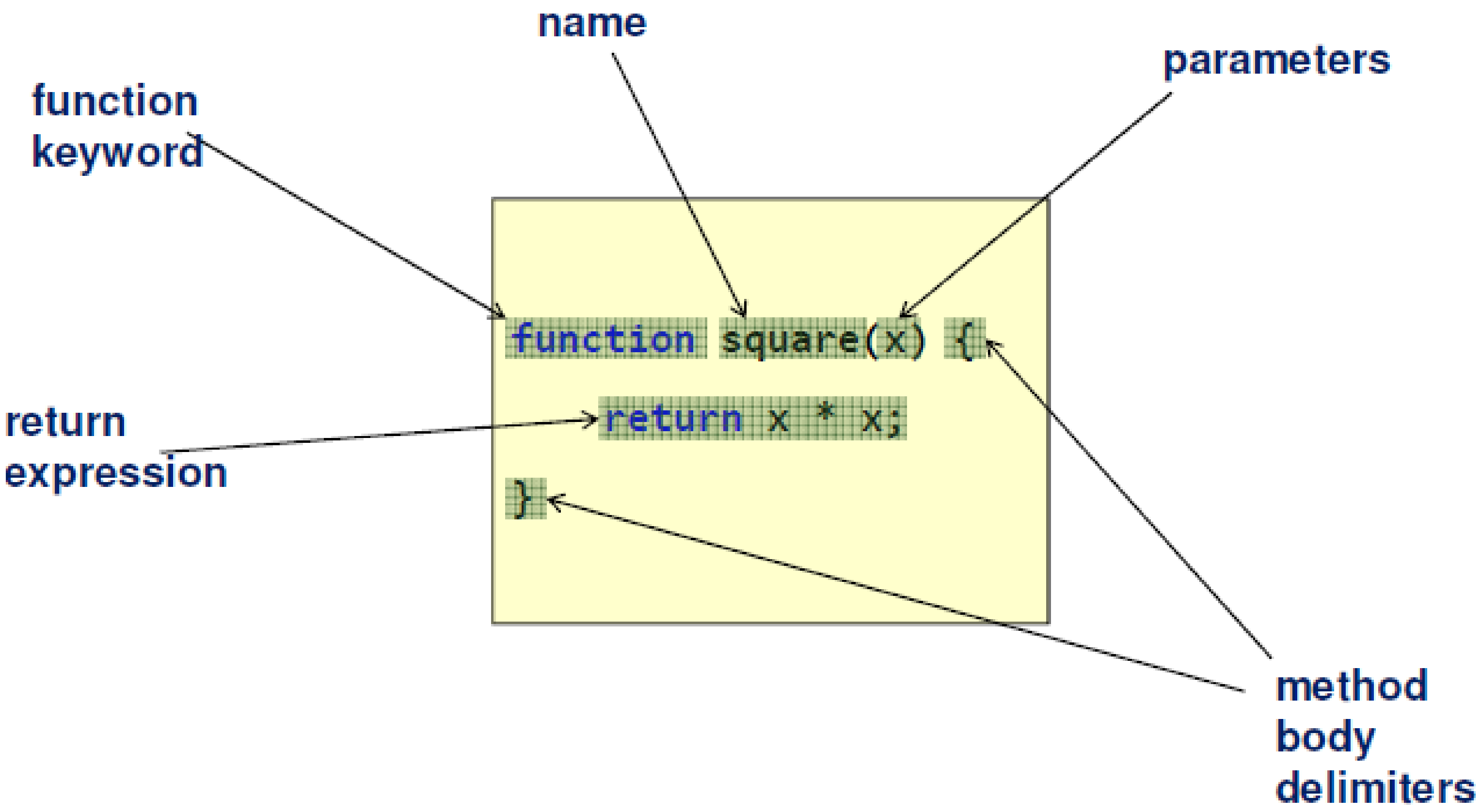
Exercise 1

- Create an object literal capable of storing the following information regarding a **hotel**:
  - ***id*** (unique identifier, integer)
  - ***name*** (string)
  - ***description*** (string)
  - ***country*** (string)
  - ***city*** (string)
  - ***addedDate*** (date)
  - ***startPrice*** (float)
- Output some of the properties to the browser console

4

Functions

Meet the function



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

name

parameters

function keyword

return expression

method body delimiters

Declaring functions

- **Standard function declaration**

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

- **Anonymous function expression**

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

Invocation

Method name followed by ()

```
square(7);
// result is 49
```

Function variable name followed by ()

```
square(7);
// result is 49
```

Declaring functions

- **Anonymous function expression**

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

- **Named function expression**

```
var square = function sqr(x) {
        return x * x;
};
```

Invocation

Function variable name followed by ()

```
square(7);
// result is 49
```

Function variable name followed by ()

```
square(7); // result is 49
sqr(7); // Error: sqr is not defined
```

Function overloading

- Functions **cannot** be overloaded
- Parameter flexibility
- Object parameters are passed by **reference**
- Primitive type parameters are passed by **value**

The arguments object

- **Local** variable available within all functions
- Contains the functions **parameters**
- Indexed like an array
- Has a length property

Recursion

- A function may call itself

```
// a recursive function calls itself
function factorial(n) {
    if (n === 0 || n === 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
factorial(5); // result is 120
```

Closure

- This leads us to one of the most **powerful** abstractions that JavaScript has to offer — but also the most potentially confusing. What does this do?

```
function makeAdder(a) {
        return function(b) {
                return a + b;
        };
}
x = makeAdder(5);
y = makeAdder(20);
x(6); // ?
y(7); // ?
```

Closure

- A closure is the combination of a function and the scope object in which it was created.
- Closures let you save **state** — as such, they can often be used in place of objects.
- An unfortunate side effect of closures is that they make it trivially easy to **leak memory**

The *new* operator

- Creates an instance of a user-defined object type or of one of the built-in object types that has a constructor function.
- A function constructor is the handle of the closest thing JavaScript has to a class
- Syntax

  new *constructor[([arguments])]*

How to create a user-defined object

1. Define the object type by writing a function:

```
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
}
```

2. Create an instance of the object with **new**:

```
var myCar = new Car("Mercedes", "C63", 2012);
```

Object.prototype

- All objects in JavaScript are descended from Object; all objects inherit methods and properties from Object.prototype
- The standard way to create an object prototype is to use an object constructor function
- The new operator simply creates new objects from the same prototype
- The prototype property allows the adding of new properties to an existing prototype:

```
Car.prototype.getDisplayText = function() {
    return this.make + ' ' + this.model + '(' + this.year + ')';
}
```

Exercise 2

- Create a **user-defined object** based on the previous hotel object literal
- Create a **prototype function** that displays the hotel name followed by the country
- Create at least two objects starting from the declared constructor function and call the previously created method.

5

Built-in types

String

- Primitive type representing an ordered set of characters
- Created using one of two literal notations:

```
var string1 = "The quick brown fox's jump";
var string2 = "'The quick brown fox'";
```

- No multiline string syntax
- Common escape sequences begin with \
- New line \n

String methods

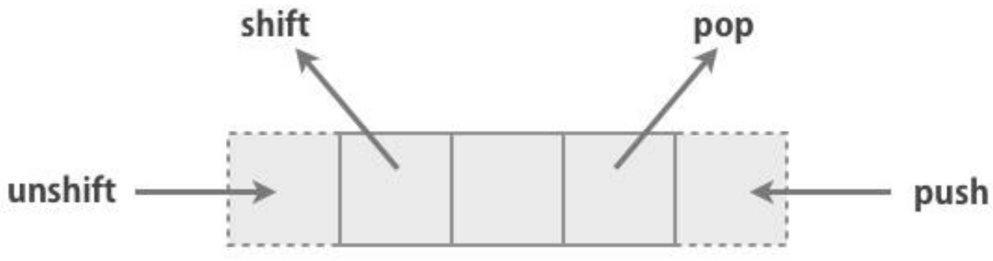
- **charAt**(index) – returns the character (as a string) at the specified position
- **indexOf**(string) – returns the index of the specified string
- **replace**(from, to) – replaces the first argument with the second argument.
- **search**(regex) – returns the index of the regex search pattern
- **slice()** – returns a substring of a string
- **split**(separator) – splits a string on separator
- **toLowerCase**()
- **toUpperCase**()

Number

- All numbers are floating point
- Standard operators +, -, *, /, %
- **toFixed**(n) – returns the number to n decimal places

Array

- An indexed collection
- Declared using the literal syntax [ ]
- Can store anything
- Many useful methods



```
var collection = ['a', 1, /3/, {}];
collection[0]; // access the first element
collection.length; // get the number of elements in the array
```

Date

- No literal syntax
- The month parameter is zero based ie. January is 0
- new Date() is the current date

```
var birthday = new Date(2010, 10, 26);
```

JSON

- JavaScript Object Notation
- Uses JavaScript object literals as a data format
- Lightweight, readable alternative to xml
- Increasingly used in AJAX web applications

JSON vs XML

```
{
        books:[
        {
                title: "Frankenstein",
                author: "Mary Shelley",
                genres: ["horror", "gothic”]
        },
        {
                title: "Moby Dick",
                author: "Herman Melville",
                genres: ["adventure", "sea"]
        }
        ]
}
```

```
<books>
        <book>
                <title>Frankenstein</title>
                <author>Mary Shelley</author>
                <genres>
                        <genre>horror</genre>
                        <genre>gothic</genre>
                </genres>
        </book>
        <book>
                <title>Moby Dick</title>
                <author>Herman Melville</author>
                        <genres>
                        <genre>horror</genre>
                </genres>
        </book>
</books>
```

Parsing JSON

There are two main recommended ways to parse JSON objects:

1. Using the Native JSON object (currently supported in Chrome, FF3.5+, IE8+, and Opera 10.5+)
2. Using **json2.js**

- Two important functions
  - **JSON.parse** – converts JSON to JavaScript objects
  - **JSON.stringify** – converts JavaScript objects to JSON

Exercise 3

- Create an ***array*** of several JavaScript objects capable of containing hotel information (you can use either **object literals** or **user-defined objects**).
- Test the following array methods: **push**, **pop**, **unshift**, **shift**
- Transform the previously created array into ***JSON*** format; output the result to the console; convert the JSON result back to a JavaScript object and visually compare the given result with the initial object.

6

Control flow and error handling

Block statement

- The most basic statement is a block statement that is used to group statements. The block is delimited by a pair of curly brackets:

```
{
  statement_1;
  statement_2;
  . .
  statement_n;
}
```

- Does **not** provider variable scope

if statement

- Execute a block if a logical condition is true

```
if (condition) {
  statement_1;
```

```
if (2 === (1 + 1)) {
        // execute this block
}
```

- *condition* can be any expression that evaluates to true or false.

if … else statement

- Alternative block to execute if the condition is false

```
if (condition) {
  statement_1;
} else {
  statement_2;
}
```

```
if (false) {

} else {
        // execute this block
}
```

switch statement

- **Choose from a set of possibilities**

```
switch (expression) {
  case label_1:
    statements_1
    [break;]
  case label_2:
    statements_2
    [break;]
    ...
  default:
    statements_def
    [break;]
}
```

```
var gender = "female";
switch (gender) {
  case "female":
    console.log("Hello, ma'am!");
    break;
  case "male":
    console.log("Hello, sir!");
    break;
  default:
    console.log("Hello!");
    break;
}
```

while statement

- Pre-tested loop
- Ensure that the loop condition will eventually become false

```
while (condition) {
        statement
}
```

```
var i = 0;
while (i < 5) {
        console.log(i);
        i += 1;
}
```

```
// Output:
// 0
// 1
// 2
// 3
// 4
```

do … while statement

- Post-tested loop
- Ensure that the loop condition will eventually become false

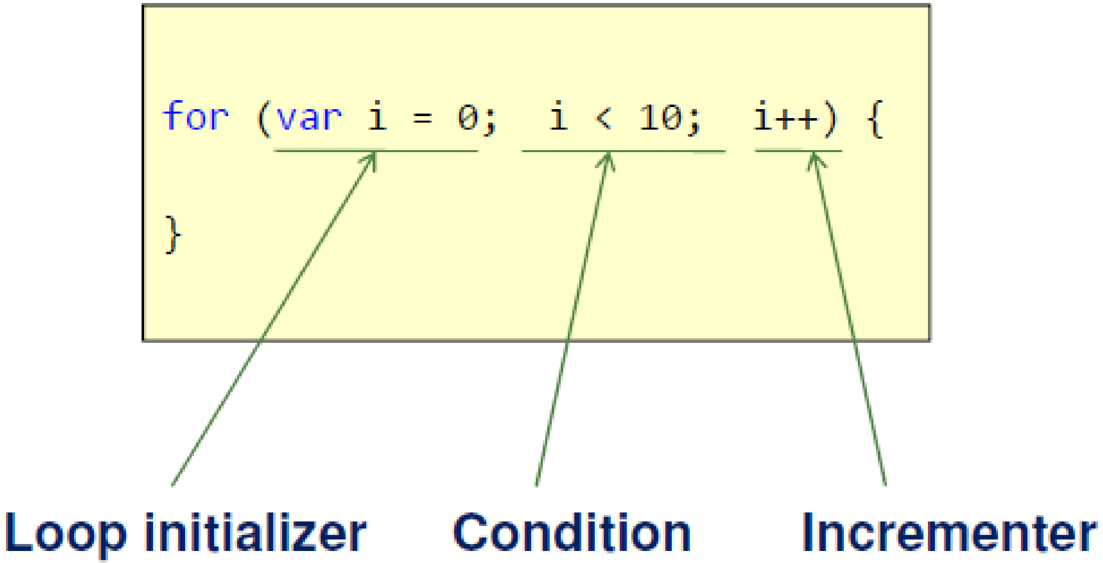
```
  do {
        statements
  } while (condition);
```

```
var i = 0;
do {
      console.log(i);
      i += 1;
} while (i < 5);
```

```
// Output:
// 0
// 1
// 2
// 3
// 4
```

teamnet

for statement



```
for (var i = 0;  i < 10;  i++) {

}
```

Loop initializer

Condition

Incrementer

for … in statement

- Iterates a specified variable over all the properties of an object.

```
for (variable in object) {
        statements
}
```

```
var obj = {a:1, b:2, c:3};
for (var prop in obj) {
        console.log("o." + prop + " = " + obj[prop]);
}
```

```
// Output:
// "o.a = 1"
// "o.b = 2"
// "o.c = 3"
```

Error handling

- Throw an exception when an unusual error condition occurs
- Exceptions are thrown using the ‘**throw**’ statement
- Just about any object can be thrown in JavaScript.
- It is frequently more effective to use one of the exception types specifically created for this purpose
- The exception object can be accessed when the exception is caught

throw statement

- Use the throw statement to throw an exception. When you throw an exception, you specify the expression containing the value to be thrown:

```
throw expression;
throw "Error2";   // String type
throw 42;         // Number type
throw true;       // Boolean type
throw {toString: function() { return "I'm an object!"; } };
```

try … catch statement

- When an exception is thrown within a ‘try’ block it can be caught and handled within a ‘**catch**’ block.
- A ‘**finally**’ block can be used to guarantee execution of some statements, even in the event of an exception

```
try {
        // statements that could generate exceptions
} catch(e) {
        // do necessary actions (eg: log exception)
} finally {
        // statements executed whether or not an exception is thrown
}
```

Exercise 4

- Starting from the previously created array of hotels, write the following:
  - Function that accepts **one argument** (a hotel object) and **adds** it to the array
  - Function that accepts **one argument** (a hotel object) and **updates** the existing hotel in the array with the new value (base on the id value)
  - Function that accepts **one argument** (a hotel id) and **removes** the corresponding hotel from the array
  - Function **without arguments** that returns the **maximum hotel id** (or null if there are no hotels)
  - Function that accepts **one argument** (a hotel id) and returns the **corresponding hotel** object from the list

*

Extra

Problems so far

- The amount of code we write is steadily increasing
- A lot of code (variables, functions) can be publicly accessed
- Namespace pollution

- We need to better organize out code

IIFE

- **Immediately-invoked function expression**
- Pronounced iffy
- JavaScript design pattern which **produces a scope** (lexical scope)
- Used to protect against polluting the global namespace

IIFE

```
(function() {
    // the code here is executed once in its own scope
}());

(function(a, b) {
    // a == 'hello'
    // b == 'world'
}('hello', 'world'));
```

IIFE

```
var counter = (function () {
    var i = 0;

    return {
        get: function () {
            return i;
        },
        increment: function () {
            return ++i;
        }
    };
} ());

// 'counter' is an object with properties
counter.get(); // 0
counter.increment(); // 1
counter.increment(); // 2
counter.get(); // 2
```

Namespace pattern

- Holds all of the user created global objects into a single object (called namespace) to prevent them from clashing with identifiers in other modules

```
var namespace = window.namespace || {};
namespace.func = function () { return 42; };
namespace.value = 123;
```

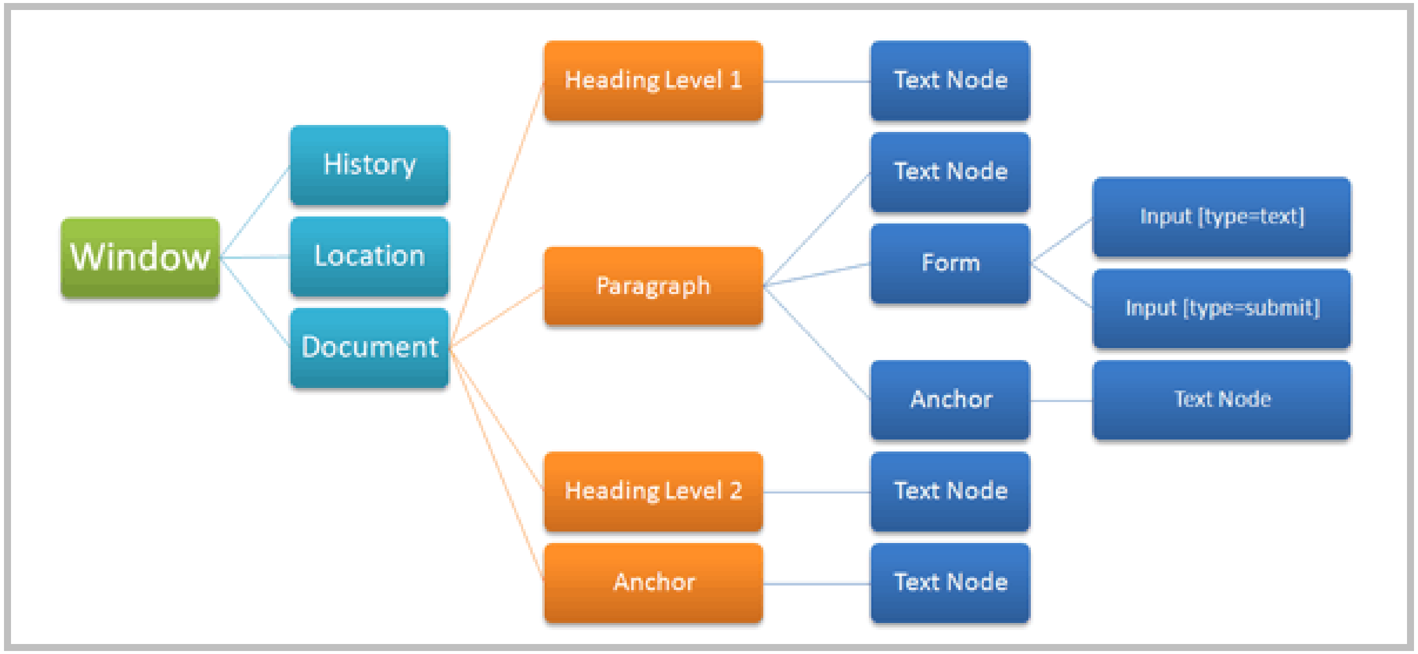
7

DOM interactions

Document Object Model (DOM)

- API for HTML, XML and SVG documents
- Provides a structured representation of the document (**tree**) as a group of nodes and objects that have **properties** and **methods**
- Nodes can also have **event handlers** attached to them
- The DOM itself is not part of the JavaScript language, though it is often accessed with it

Document Object Model (DOM)



DOM Methods

Old methods:

- **document.getElementById(id)** – returns a reference to the DOM node having the given id
- **document.getElementsByTagName(tag)** – returns a live node list based on the given tag (eg: div, ul, li, …)

Relatively new methods (IE 8+):

- **document.querySelector(selector)** – returns the first element within the document that matches the specified selector
- **document.querySelectorAll(selector)** – returns a list of elements that match the specified selector

Exercise 5

Go to http://jsfiddle.net/jvbheqkp/

Select the following:

- The node having the id “hotelsContainer”
- All the span tags that are children of the node with the id “third”
- All the nodes having the class “right”

Questions?

teamnet transforming technology

Thank you!