Javascript

André Restivo

Index

Introduction

Javascript

- Javascript is a **prototype-based**, **dynamic**, **object-oriented**, **imperative** and **functional** language.
- In Javascript, functions are considered **first-class** citizens.
- Most commonly used as part of web browsers as a **client-side** scripting language.

History

- Originally developed by **Brendan** Eich at **Netscape**.
- Developed under the name **Mocha** but later named **LiveScript**.
- Changed name from LiveScript to **JavaScript**, in **1995**, at the time Netscape added support for Java.
- Microsoft introduced JavaScript support in Internet Explorer in August **1996** (called JScript).
- Submitted to **Ecma** International for consideration as an industry standard in 1996 (**ECMAScript**).
- Ecma International released the first version of the specification in **1997**.
- Nowadays JavaScript is a trademark of the **Oracle** Corporation.
- But JavaScript is officially managed by the **Mozilla** Foundation.

Console

- Modern browsers all have a Javascript console that can be used to log messages from within web pages.
- It can also be used to inspect variables, evaluate expressions and just plain experimentation.
- The specifics of how it works vary from browser to browser, but there is a de facto set of features that are typically provided.
- The **console.log(msg)** function outputs a message to the console.
- Other debug level are possible like **console.info(msg)**, **console.warn(msg)** and **console.error(msg)**.
- Browsers allow filtering messages depending on their level.

Alert

The alert function opens a popup window with some text.

```
alert("Hello world!");
```

Resources

- Reference:
    - MDN Javascript Reference
    - EcmaScript Reference
    - MDN DOM Reference
- Resources:
    - MDN Javascript Resources
    - JS Fiddle
- Tutorials:
    - jQuery: Javascript 101

Variables

Variables

- JavaScript is a loosely typed or a dynamic language. That means you don't have to declare the type of a variable ahead of time.
- The type will get determined automatically while the program is being processed.
- Variables are declared using the **var** instruction:

```
var foo = 10;
foo = 'John Doe';
foo = true;
```

Primitive Data Types

The ECMAScript standard defines the following data types:

- Boolean (**true** or **false**)
- Null (only one possible value: case sensitive **null**)
- Undefined (**not** been **assigned** a value)
- Number (**double**-precision 64-bit)
- String (**text**ual data - single or double quoted)

The + Operator

The plus (+) operator sums numbers, but if one of the operands is a string, it converts the other one into a string and concatenates the two:

```
console.log(11 + 31);   // 42
console.log("11" + 31); // "1131"
console.log(11 + "31"); // "1131"
```

Control Structures

If ... else

- Use the **if** statement to execute a statement if a logical condition is true.
- Use the optional **else** clause to execute a statement if the condition is false.

```
if (condition) {
  //do domething
} else {
  //something else
}
```

Boolean evaluation

The following values all evaluate to false:

- false
- undefined
- null
- 0
- NaN (not a number)
- the empty string

All other values, including objects evaluate to true.

Be careful with the Boolean object:

```
var foo = new Boolean(false);
if (foo) // evaluates to true
```

Equality

- Strict equality compares two values for equality.
- Neither value is implicitly converted to some other value before being compared.
- If the values have different types, the values are considered unequal.

```
0 === 0     // true
0 === "0"   // false
0 === false // false
```

- Loose equality compares two values for equality, after converting both values to a common type.

```
0 == 0     // true
0 == "0"   // true
0 == false // true
```

Switch

- A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label.
- If a match is found, the program executes the associated statement.

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

Loops

JavaScript supports the **for**, **do while**, and **while** loop statements:

```
for (var i = 0; i <= 10; i++) {
  console.log(i);
}// 0 1 2 3 4 5 6 7 8 9 10
```

```
var i = 0;
do {
   console.log(i);
   i++;
} while (i <= 10); // 0 1 2 3 4 5 6 7 8 9 10
```

```
var i = 0;
while (i <= 10) {
   console.log(i);
   i++;
} // 0 1 2 3 4 5 6 7 8 9 10
```

Break and continue

- The break statement finishes the current loop prematurly.
- The continue statement finishes the current iteration and continues with the next.

```
for (var i = 0; i < 10; i++) {
  if (i == 8) break;
  if (i % 2 == 0) continue;
  console.log(i);
} // 1 3 5 7
```

Functions

Defining functions

A function is defined using the **function** keyword.

```
function add(num1, num2) {
  console.log(num1 + num2);
}
add(1, 2);
```

- Primitive parameters are passed to functions by value.
- Non-primitive parameters (objects) are passed by reference.

Return

Functions can also return values.

```
function add(num1, num2) {
  return num1 + num2;
}
console.log(add(1,2));
```

Objects

Objects

- JavaScript is designed on a simple **object-based** paradigm.
- An object is a collection of **properties**, and a property is an association between a name and a value.
- A property's value can be a function, in which case the property is known as a **method**.
- JavaScript is a **prototype-based** language and **does not** have a class statement.

```
var person = new Object();
person.name = "John Doe";
person.age = 45;
```

Objects as Arrays

- Properties of JavaScript objects can also be accessed or set using a bracket notation.
- Objects can be seen as associative arrays, since each property is associated with a string value that can be used to access it.

```
var person = new Object();
person["name"] = "John Doe";
person["age"] = 45;
```

For ... in

- The **for...in** statement iterates a specified variable over all its properties.
- For each distinct property, JavaScript executes the specified statements.

```
for (var foo in person)
  console.log(foo + " = " + person[foo]);
```

Almost Everything is an Object

- In JavaScript, almost everything is an object.
- All primitive types except null and undefined are treated as objects.

```
var name = "John Doe";
console.log(name.substring(0,4));
```

- In this example, the primitive type is casted temporarily into a String object that is discarded afterwards.

Object Initializer

- Object initializers can be used to create objects.
- Objects can contains other objects.

```
var person = {name: "John Doe",
              age: 45,
              car : {make: "Honda", model: "Civic"}
             };
console.log(person);     // Object {name: "John Doe", age: 45, car: Object}
console.log(person.car); // Object {make: "Honda", model: "Civic"}
```

Methods

- Methods are properties of an object that happen to be functions.
- Methods are defined the way normal functions are defined, except that they are assigned as the property of an object.
- You can use the **this** keyword within a method to refer to the current object.

```
var person = {name: "John Doe",
              age: 45,
              car: {make: "Honda", model: "Civic"},
              print: function() {
                console.log(this.name + " is " + this.age + " years old!");
              }
             };
person.print(); // John Doe is 45 years old!
```

Assigning methods

Methods can be assigned to objects just like properties.

```
var person = {name: "John Doe",
              age: 45,
              car: {make: "Honda", model: "Civic"},
             };
person.print = function() {
  console.log(this.name + " is " + this.age + " years old!");
}
person.print(); // John Doe is 45 years old!
```

Getter and Setters

- A **getter** is a method that gets the value of a specific property.
- A **setter** is a method that sets the value of a specific property.

```
var person = {
    firstName: 'John',
    lastName: 'Doe',
    get fullName() {
        return this.firstName + ' ' + this.lastName;
    },
    set fullName (name) {
        var words = name.toString().split(' ');
        this.firstName = words[0] || '';
        this.lastName = words[1] || '';
    }
}

person.fullName = 'John Doe';
console.log(person.firstName); // John
console.log(person.lastName)   // Doe
console.log(person.fullName)   // John Doe
```

Constructor functions

Functions can be used to create new objects using the **new** keyword.

```
function Person (name, age, car) {
  this.name = name;
  this.age = age;
  this.car = car;
  this.print = function() {
    console.log(this.name + " is " + this.age + " years old!");
  }
}

var john = new Person("John Doe", 45, {make: "Honda", model: "Civic"});
person.print(); // John Doe is 45 years old!
```

Functions are Objects

When a function is created using the **function** keyword we are really defining an object.

```
function sayHello() {
  console.log("Hello");
}

sayHello();                                    //Hello
sayHello.info = "This function says hello!";

console.log(sayHello.info);                    //This functions says hello!

sayHello.goodBye = function() {
  console.log("Goodbye");
}

sayHello();                                    //Hello
sayHello.goodBye();                            //Goodbye
```

Prototype

- Each Javascript function has an internal **prototype** property that is initialized as a nearly empty object.
- When the **new** operator is used on a constructor function, a new object is created that has the same prototype as the constructor function. The function is then executed having the new object as its context.
- We can change the prototype of a function by changing the **prototype property** directly.

```
function Person(name) {
  this.name = name;
}

var john = new Person("John Doe");
Person.age = 45;                    // Only changes the Person object
                                    // not its prototype.

var jane = new Person("Jane Doe");
console.log(jane.age); // undefined

Person.prototype.age = 45;          // Changes the prototype.
var mary = new Person("Mary Doe");  // All objects constructed using the
console.log(mary.age); //45         // person constructor now have an age.
console.log(john.age); //45         // Even if created before the change.
```

Prototype

You can inspect the prototype of a function easily in the console.

```
function Person(name) {
  this.name = name;
}

Person.prototype; // Person {}
Person.prototype.saySomething = function (){console.log("Something")};
Person.prototype; // Person {saySomething: function}

var john = new Person();
john.saySomething()        // Something
john.constructor;          // function Person(name) { this.name = name; }
john.constructor.prototype // Person {saySomething: function}
```

Object __proto__

When a object is created using **new**, a **__proto__** property is initialized with the prototype of the object.

```
function Person(name) {
  this.name = name;
}

Person.prototype.saySomething = function (){console.log("Something")};
var john = new Person("John");
john.prototype; // undefined
john.__proto__; // Person {saySomething: function}
```

Call

The call method of a function (object), calls that function with a context passed as a parameter.

```
function printGreeting(greeting) {
  console.log(greeting + " " + this.name);
}

printGreeting("Hello");             // Hello [object Object]

var john = {name: "John"};
printGreeting.call(john, "Hello");  // Hello John
```

Inheritance

- Inheritance can be emulated in Javascript by changing the prototype chain.
- Every time a property is accessed on an object, if the object doesn't have that property, Javascript will lookup it up in the **__proto__** property chain.

```
function Person(name) {
  this.name = name;
}

Person.prototype.print = function() {console.log(this.name);}

function Worker(name, job) {
  this.job = job;
  Person.call(this, name);
}
Worker.prototype = new Person;
Worker.prototype.print = function() {console.log(this.name + " is a " + this.job);}

var mary = new Person("Mary");
mary.print(); // Mary
var john = new Worker("John", "Builder");
john.print(); // John is a Builder
```

Arrays

Arrays

- Arrays are **list-like objects** whose prototype has methods to perform traversal and mutation operations.
- JavaScript arrays are zero-indexed
- Arrays can be initialized using a bracket notation:

```
var years = [1990, 1991, 1992, 1993];
console.log(years[0]); // 1990
years.info = "Nice array";
console.log(years.info); // Nice array
```

Array elements are object properties but they cannot be accessed using the **dot** notation because their name is not valid.

```
var years = [1990, 1991, 1992, 1993];
console.log(years[0]); // 1990
console.log(years.0);  // Syntax error
```

Array prototype

By changing the Array prototype we can add methods and properties to all arrays.

```
var years = [1990, 1991, 1992, 1993];
Array.prototype.print = function() {
  console.log("This array has length " + this.length)
};
years.print();
```

Array prototype methods

These are some of the methods defined by the Array prototype:

- Properties: prototype, length
- Mutators: fill, pop, push, reverse, shift, sort, splice, unshift
- Accessor: concat, contains, join, slice, indexOf, lastIndexOf
- Iterator: forEach, entries, every, some, filter

Some examples:

```
var years = [1990, 1991, 1992, 1993];
years.push(1994);
console.log(years.length); // 5

years.reverse();
console.log(years);        // [1994, 1993, 1992, 1991, 1990]

var sum = 0;
years.forEach(function (element, index, array) {sum += element});
console.log(sum);          //9960

years.every(function (element, index, array) {return element >= 1990});
years.some(function (element, index, array) {return element % 2 == 0});
```

Exceptions

Throw

- You can throw exceptions using the **throw** statement.
- You can throw any expression.

```
function UserException (message){
  this.message=message;
  this.name="UserException";
}

UserException.prototype.toString = function (){
  return this.name + ": " + this.message;
}

throw new UserException("Value too high");
```

```
throw "This is an error";
```

Error Object

If you are throwing your own exceptions, in order to take advantage of the name and message properties, you can use the **Error** constructor.

```
throw new Error("This is an Error");
```

Try ... Catch

The **try...catch** statement marks a block of statements to try, and specifies a response, should an exception be thrown.

```
try {
  // code to try
}
catch (e) {
   // statements to handle any exceptions
}
```

DOM

DOM

- The **Document Object Model** (DOM) is a **programming interface** for HTML and XML documents.
- It provides a structured representation of the document and it defines a way that the structure can be accessed from programs so that they can change the document **structure**, **style** and **content**.
- The DOM is a fully object-oriented representation of the web page, and it can be modified with a scripting language such as **JavaScript**.

Javascript on HTML Documents

Javascript can be embeded directly into an HTML document:

```
<script>
  // javascript code goes here
</script>
```

Or as an external resource:

```
<script type="text/javascript" src="script.js"></script>
```

Script tag position

As Javascript is capable of changing the HTML structure of a document, whenever the browser finds a **script** tag, it first fetches and runs that script and only then resumes loading the page.

Most Javascript scripts don't change the document until it is fully loaded but the browser does not know this. For that reason, it was recommended that **script** tags were placed at the bottom of the **body**.

Modern browsers support the async and defer attributes, so scripts can safely be placed in the **head** of the document:

```
<head>
  <script type="text/javascript" src="script.js" async></script>
  <script type="text/javascript" src="script.js" defer></script>
</head>
```

- A asynchronous (**async**) script is run as soon as it is downloaded but without blocking the browser.
- Deferred (**defer**) scripts are executed only when the page is loaded and in order.

Document

- The Document object represents an HTML document.
- You can access the current document in Javascript using the **global** variable **document**.

Some Document **methods**:

Element **getElementById**(id)	returns the element with the specified id
NodeList **getElementsByClassName**(class)	returns all element with the specified class
NodeList **getElementsByTagName**(name)	returns all element with the specified tag name
Element **createElement**(name)	creates a new element.

```
var menu = document.getElementById("menu");
var paragraphs = document.getElementsByTagName("p");
```

Some Document **properties**: URL, title, location

Element

An Element object represents an HTML element.

Some common Element **properties**:

id	The id attribute
innerHTML	The HTML code inside the element
outerHTML	The HTML code including this element
style*	The CSS style of the element

Some common Element **methods**:

String **getAttribute**(name)	get the attribute with the given name (or null).
setAttribute(name, value)	modifies the attribute with the given name to value.
remove()	removes the element from its parent.

Other **methods**: **removeAttribute**, **hasAttribute**

HTML Element

The HTMLElement inherits from the Element object. There are different HTMLElement objects for each HTML element.

HTMLElement	style, title, blur(), click(), focus()
HTMLInputElement	name, type, value, checked, autocomplete, autofocus, defaultChecked, defaultValue, disabled, min, max, readOnly, required
HTMLSelectElement	name, multiple, required, size, length
HTMLOptionElement	disabled, selected, defaultSelected, text, value
HTMLAnchorElement	href, host, hostname, port, hash, pathname, protocol, text, username, password
HTMLImageElement	alt, src, width, height

Node

The Node object represents a node in the document tree. The Element object inherits from the Node object.

Some common Node **properties**:

firstChild and **lastChild**	first and last node child of this node.
childNodes	all child nodes as a NodeList.
previousSibling and **nextSibling**	previous and next sibling to this node.
parentNode	parent of this node.
nodeType	not all nodes are elements: see Node type list

Some common Node **methods**:

appendChild(node)	appends a node to this node.
replaceChild(new, old)	replaces a child of this node.
removeChild(child)	removes a child from this node.
insertBefore(new, reference);	inserts a new child before the reference child.

Element and Node

Some examples:

```
var element = document.getElementById("menu"); // gets the element with id menu

element.style.color = "blue";                  // changes the text color to blue
element.style.padding = "2em";                 // and the padding to 2em

var paragraph = document.createElement("p");   // creates a new paragraph
paragraph.innerHTML = "Some text";             // inserts text in the paragraph

element.appendChild(paragraph);                // adds the paragraph to the menu
element.remove();                              // removes the menu
```

NodeList

- A Node List is an array of elements, like the kind that is returned by the method **document.getElementsByTagName()**.
- Items in a Node List are accessed by index like in an array:

```
var elements = document.getElementsByTagName("p");
for (var i = 0; i < elements.length; i++) {
  var element = elements[i];
  // do something with the element
}
```

Events

- Events are sent to notify code of interesting things that have taken place.
- Each event is represented by an object which is based on the Event interface, and may have additional custom fields and/or functions used to get additional information about what happened.

Some possible events:

Mouse	click, dblclick, mousedown, mouseup, mouseenter, mouseleave, mouseover, mousewheel
Keys	keypress, keydown, keyup
Text	cut, copy, paste, select
Form	reset, submit
Input	focus, blur, change

Events in HTML

A possible way to get notified of Events of a particular type (such as click) for a given object is to specify an event handler using:

An HTML attribute named on{eventtype} on an element, for example:

```
<button onclick="return handleClick(event);">
```

or by setting the corresponding property from JavaScript, for example:

```
document.getElementById("mybutton").onclick = function(event) { ... };
```

Add Event Handler

On modern browsers, the Javascript function **addEventListener** should be used to handle events.

```
element.addEventListener(type, listener[, useCapture])
```

useCapture: If true, useCapture indicates that the user wishes to initiate capture. All events of the specified type will be dispatched to the registered listener before being dispatched to any target beneath it in the DOM tree.

Example:

```
function handleEvent() {
  ...
}

var menu = document.getElementById("menu");
menu.addEventListener("click", handleEvent, false);
menu.addEventListener("click", function(){...}, false);
```

The **this** keyword can be used to access the element where the event was triggered.

Event Handler Functions

A function that handles an event can receive a parameter representing the event that caused the function to be called.

```
function handleEvent(event) {
  ...
}

var menu = document.getElementById("menu");
menu.addEventListener("click", handleEvent, false);
```

Depending on its type, the event can have different properties and methods: Reference

Ajax

Ajax

- Asynchronous JavaScript + XML,
- Not a technology in itself, but a term coined in 2005 by **Jesse James Garrett**, that describes an approach to using a number of existing technologies: namely the **XMLHttpRequest** object.

XMLHttpRequest

XMLHttpRequest makes sending HTTP requests very easy.

```
void open(method, url, async);
```

Example:

```
function requestListener () {
  console.log(this.responseText);
}

var request = new XMLHttpRequest();
request.onload = requestListener;
request.open("get", "getdata.php", true);
request.send();
```

Monitoring Progress

```
var request = new XMLHttpRequest();

request.addEventListener("progress", updateProgress, false);
request.addEventListener("load", transferComplete, false);
request.addEventListener("error", transferFailed, false);
request.addEventListener("abort", transferCanceled, false);

request.open("get", "getdata.php", true);
request.send();

function updateProgress (event) {
  if (event.lengthComputable)
    var percentComplete = event.loaded / event.total;
}

function transferComplete(event) {
  alert("The transfer is complete.");
}

function transferFailed(event) {
  alert("An error occurred while transferring the file.");
}

function transferCanceled(event) {
  alert("The transfer has been canceled by the user.");
}
```

Analyzing a XMLRequest Response

If you use XMLHttpRequest to get the content of a remote **XML** document, the responseXML property will be a DOM Object containing a parsed XML document, which can be hard to manipulate and analyze.

If you use **JSON**, it is very easy to parse the response as JSON is already in Javascript Object Notation.

```
JSON.parse('{}');              // {}
JSON.parse('true');            // true
JSON.parse('"foo"');           // "foo"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse('null');            // null
JSON.parse('{"1": 1, "2": 2}') // Object {1: 1, 2: 2}
JSON.parse(this.responseText)  // The server response
```

Wat

https://www.destroyallsoftware.com/talks/wat