Seminar Report

JavaScript

Ivan Hell
Ivan.Hell@student.uibk.ac.at

15 February 2013

Supervisor: Cezary Kaliszyk

Abstract

This paper is about the programming language JavaScript. It tries to give the reader a brief glimpse of the history and the evolution of the language. Step by step, the basic concepts are discussed. In this context, practical examples are used to always have a relation to the practice.

Contents

1 Introduction

This paper is about JavaScript, a scripting language that today mainly is known as language for the client-side manipulation of websites. The paper tries to give a short introduction in the concepts and ideas behind the language. To understand it, the reader should have a basic knowledge about programming languages like C or Java, because discussing all details would go beyond the scope of this work. The paper first talks about the history of the language, how it became what it is today. After that, some basic concepts will be discussed, that should enable the reader to read and write small pieces of JavaScript-code on his own. In the end I will then talk about the application areas of the language, for what it can be used and the main ideas behind these use cases.

2 History

Java Script is a scripting language which originally was developed by Brendan Eich, an employee of the company *Netscape Communications*, in 1995. At the beginning, the language was called LiveScript [3]. After an agreement with the company *Sun Microsystems* it was renamed to JavaScript. Although the name JavaScript might suggest, that it is closely related to the programming language *Java*, this is not the case, because these two languages have many fundamental concepts, which differ from each other.

JavaScript was developed as a feature for the web-browser *Netscape Navigator 2.0*, which allowed to influence and modify HTML-pages dynamically at the client-side. With this new extension, for example the browser should have been able to verify form-inputs, before sending them back to the host. With the beta version of the *Netscape Navigator 3.0*, Netscape introduced version 1.1 of JavaScript that offered much more possibilities for the so called *dynamic HTML*. With this new version, it was also possible to have *rollover-graphics*, a picture that changes, if the user moves the mouse-pointer over the image.

At the same time, Microsoft invented the new script-language for its own browser too. With *Internet Explorer 3* Microsoft presented a browser with JScript support, a proprietary script language very similar to JavaScript. From this point on, the two companies always expanded the script languages. In 1997 the ECMA [1] released, in cooperation with *Netscape Communications*, the ECMA-262 standard, which used to describe the basic concept and structure of the script language. This standard is now available at version 5.1, that was released in 2011 [2]. The latest stable JavaScript version 1.8.6, that has full support for this version of the standard [2]. Microsoft's alternative scripting language JScript, was released in version 9.0 and is based on the fifth edition of the the standard.

Although JavaScript is mainly used for client-side manipulation of web-pages, there also exist several projects, which allows to use it as a server-side scripting language. In this case, the script will not be executed in a web-browser as usual,

[1] **E**uropean **C**omputer **M**anufacturers **A**ssociation and today Ecma Internationa

[2] Can be downloaded here

but on the server itself.

Nowadays JavaScript is mainly used to create embedded scripts in HTML-pages, that are used to animate web-sites, to validate user-inputs, to control the browser, to load data dynamically or to implement a window-management system using pop-up windows.

3 Variables and data-types

When writing an imperative program, it is always necessary to hold some data, so that it can be used later again. For this reason there exist variables. These variables, in JavaScript can be defined using the keyword *var* in front of a variable name. Using this keyword, a fresh and empty variable will be created, if it was not defined previously (otherwise nothing will happen). The name of the variable can be an arbitrary sequence of characters, numbers and underscores which, however, can not begin with a number [5]:

```
var bottle; //valid definition
var _BE3Er; //valid definition
var heL lo; //invalid definition (blank)
var 8-test; //invalid definition (leading number, hyphen)
```

Further, JavaScript has the property that it is a case-sensitive language [1]. In order to this, `var Beer;` and `var beer;` are two different variable definitions.

Each variable, after its declaration, can be associated with any data. To determine the type of the data, the function *typeof(data)* can be used. In JavaScript it is possible be distinguished between six basic data-types: Number, String, Boolean, Object, Null and Undefined (no value was assigned to the variable) [4].

3.1 Number

Like in almost all programming-languages, JavaScript also supports the use of numbers. The big difference to other languages is, that in JavaScript there only exists a unique data-type for floating point and integer numbers. For this reason, internally all numbers are represented as a 64-bit floating point number, according to the IEEE 754 standard. Both the minimal negative and maximal positive number, that can be represented in JavaScript, are defined in `Number.MAX_VALUE` and `Number.MIN_VALUE`.

3.2 String

To represent a sequence of characters, the data-type String is used. The maximum length of such a string is not specified in the ECMA-standard, so that it depends on the JavaScript interpreter, that is used to execute the script.

In general, strings are defined using quotation marks. Single or double quotes can be used, where the same quotes must be used to open and close the string. This has the advantage, that it is possible to use single quotes within a string, defined with double quotes, and vice versa:

```
var var1 = "Bottles"; //valid definition
var var2 = 'Be"er';   //valid definition
var var3 = "Test';    //invalid definition because of
                      //different quotation marks
```

Like in many other programming-languages, also in JavaScript the *plus* operator can be used to concatenate two strings:

```
var word1 = "Hello ";     //defining normal string
var word2 = "world"+"!";  //defining concatenated string
var result = word1+word2; //concatenating both strings
                          //with the result "Hello World!"
```

3.3 Boolean

In order to deal with numbers, that represent truth-values, JavaScript provides the data-type Boolean. Variables of this type can only be assigned the values `true` or `false`.

4 Basic concepts

Like in most other scripting languages, JavaScript's typing is dynamic. Therefore variables do not have an explicit type. That means, a variable x, that earlier was assigned for example a number, later could be assigned a Boolean or an object. Nevertheless, each generated object has a type, that is checked during the execution of the program. Further, JavaScript is also weak typed, which means that the interpreter is able to automatically cast data of a given type into another type. This is performed using the following rules: A string can be converted to a number, if the content consist of a valid number (empty string is mapped to number zero); It is possible to convert a Boolean to its number values (`true=1` and `false=0`) if it is necessary. For this reasons it is possible to subtract a string (has to be a number, defined as string) from a number. But this property is not always an advantage. For example the transitivity of the equals operator is no longer given (while the statements `"" == 0` and `0 == "0"` both evaluate to true, `"" =="0"` is not a true statement).

In addition, JavaScript is a multi-paradigm programming language, which offers the possibility to write programs in three different styles:

- Object-oriented programming:

  In JavaScript everything is represented as an object. Each object has its own properties and functions which are objects too. Inheritance in JavaScript is realized using prototypes which will be explained later.

- Imperative programming:

  Like many other programming languages, JavaScript also provides several control structures like loops, if-than-else statements, ... For this reason, programs can be defined as a sequence of commands, which is the main characteristic of a imperative program.

- Functional programming:

  Because of the fact, that in JavaScript each function is an individual object, it is possible to pass a function as an argument to another function. Therefore, functions are able to have other function as its return value. Therefore, JavaScript can also be used as a functional programming language.

4.1 Notation

In JavaScript the properties of an object can be accessed in two different ways:

array-notation: The property of the object is called by the object's name, followed by the property's name in square brackets (property's name is represented as a string-object). For example the instruction `bottle["content"]` would access the property *content* of the object *bottle*.

dot-notation: The property of the object is called by the object's name, followed by a dot and the property's name (as known from other languages like Java, C#, ...). Hence, the instruction `bottle.content` is equivalent to `bottle["content"]`.

4.2 Functions in JavaScript

As said before, each function in JavaScript can be seen as full-fledged object with its own properties and sub-functions which, however, are only accessible from within the function itself. It is possible to distinguish between two types of functions: named-functions and anonymous-functions. A named function can easily be created by using the *function* keyword:

```
function printBottle(param){
    document.write("Hello bottle from " + param + "!");
}
printBottle("world"); //calling the function
```

In this example a function named *printBottle* will be created, that accepts one parameter. Calling this function with the parameter `world` would cause the script to print the words `Hello bottle from world`. A further possibility to define this function is to declare it as an anonymous-function:

```
var func = function (param){
    document.write("Hello bottle from " + param + "!");
}
func("world"); //calling the function
```

Unlike in the previous example, the defined function has no name. Now the property is used that each function is an object that can be assigned to any variable. In this case, the function was assigned to the variable *func*, by which it can be called as in the previous example.

4.3 Object-handling

Differently from conventional object-oriented languages, JavaScript does not provide classes. Usually these classes are of use as a template, which is used to instantiate new objects. In JavaScript, a new object can be created using the *new* keyword which is followed by a so called constructor-function, that is used to initialize the new object. JavaScript already provides some predefined constructor-functions like *Object()*, *String(param)*, ... Nevertheless it is also possible to define custom constructor-functions, in which the *this* keyword can be used, to access the object's properties and functions:

```
function BeerBottle(content){
    this.content = content;
    this.drink = function(){
        this.content = "empty";
    }
}
var duffBeer = new BeerBottle("Duff");
// sets content to empty
duffBeer.drink();
```

As shown in this example the constructor-function *BeerBottle(content)* is used to create a new object that provides a property named *content* and a function *drink()*. After calling the *drink()* function the property *content* of the object will be set to `empty`.

An other way to create an object with the same properties could be, to create a new object using the constructor-function *Object()* and set the properties manually:

```
var duffBeer = new Object();
duffBeer.content = "Duff";
duffBeer.drink = function(){
    this.content = "empty";
}
// sets content to empty
duffBeer.drink();
```

4.4 Inheritance using prototypes

In JavaScript, inheritance is realized using so called prototypes. In general, each object has got a default set of properties or functions. One of this properties is called *prototype*. It is possible to bind a reference to an other object on this property. From that point on, this object is used as *super-object*. If now an object does not provide a given property, the script-interpreter searches for the property in the super-object. Because of this recursive definition, it is possible to build long lists of inheritance. To create a new object that should inherit properties from an other object, the prototype field of the constructor function has to be set (it is impossible to change the prototype of an existing object).

New created objects "inherit" all properties and functions from the prototype, so that only the new one has to be defined:

```
function Bottle(){
    this.capacity = "0.5 l"
}
function BeerBottle(content){
    this.content = content
}
BeerBottle.prototype=new Bottle();

var duffBottle = new BeerBottle("Duff");
// prints Content: Duff
document.write("Content: "+duffBottle.content+"<br>");
// prints "Capacity: 0.5 l"
document.write("Capacity: "+duffBottle.capacity+"<br>");
```

In this case, the prototype of the constructor-function *BeerBottle(content)* was set to an object that was created, using the *Bottle()* constructor-function. Thereby, all objects that are created using this function, have the property *content* as well as *capacity* which was "inherited" from the prototype object.

5 Control structures

5.1 Case distinctions

In imperative programming languages, the flow of a program is always handled by case distinctions. Therefore a program can change its behavior at a certain point, based on decisions that were taken during the runtime. To do so, JavaScript provides two concepts, that are well-known from other languages like Java, C++, ...:

- **if-then-else**

  A certain expression is evaluated. If this expression evaluates to true, a given code-section will be executed. Otherwise the code of the else-section is handled. Apart from mathematical checks ($<$, $>$, $<=$, $>=$, $==$, ...), the expression can be a function with a Boolean-object as return-value too. Another operator that often is used in this context is the $===$ operator, that checks the equality of two objects, based on their content and type (no automatic type conversation is done) [1]. Therefore `1=="1"` is a true statement, while `1==="1"` evaluates to false.

- **switch-case**

  A certain base-object is compared with other objects using the $===$ operator. If an object is equals the base-object, the code-section that was assigned to this object is executed:

```
switch (bottle.content){
    case "empty":
        bottle.fill();
        break;
    case "full":
        bottle.drink();
        break;
}
```

In this example the property *content* of the object *bottle* is compared against the string-objects `empty` and `full`. In both cases, a various code-section is executed. In difference to other programming languages, for example Java, with this switch-case-statement all data-types can be handled.

5.2 Loops

Sometimes it is necessary, to execute a code-section several times in a row. For this reason, JavaScript provides different kinds of loops that can be classified as followed:

- **while-loop**

  The while loop, the simplest form of a loop, repeats a section of code until a certain condition does no longer hold. This condition will be checked at the beginning of each iteration of the loop:

  ```
  // initializing variable bottles
  var bottles = 99;
  // repeat loop until values of
  // bottles is less than 0
  while ( bottles > -1 ) {
      // print current value of variable bottles
      document.write(bottles+" bottles on the wall.<br>");
      // decreasing variable bottles by one
      bottles = bottles - 1;
  }
  ```

  In this example the variable bottles is initialized with the number 99. Processing the *while*-loop, this number is repeatedly decreased until its value is -1. At this point the loop will be stopped and the rest of the program is processed. In order to this the output of the script would be a hundred times the sentence `X bottles on the wall.`, where `X` stands for the current value of the variable bottles.

- **do-while-loop**

  As the *while*-loop is repeated as long as the given condition holds, it is also possible that the loop is never executed (condition does not hold, when the loop is executed for the first time). For this reason, in JavaScript we can use a so called *do-while*-loop. This loop will be executed at least

once, because the abort-condition is always checked at the end of each iteration:

```
// defining invalid condition
var condition = false;
do {
    // executing loop once
    document.write("Hello beer.");
// abort loop because condition does not hold
} while (condition == true);
```

Although the variable *condition* in this example was initialized with `false` and the loop will only be executed if it is set to `true`, the loop will be executed once. This will cause the script to print out `Hello beer.`, because the condition is checked for the first time at the end of the first iteration of the loop.

- **for-loop**

  Another possibility to repeat a code section several times in a row, is to use the *for*-loop, which is of the syntax:

  ```
  for ( inits; condition; commands ) {
      // code here
  }
  ```

  inits: This section will be executed before the loop is processed for the first time. Generally it is used to initialize control-variables that will stop the loop after a given number of iterations. It is also possible to have multiple instruction in this section, that are all separated by a comma. For that t should be noted, that variable definitions using `var` keyword cannot be mixed up with other instructions (either variable definitions, or variable initializations mixed with commands).

  condition: The condition that will be checked at the beginning of each iteration. The loop is repeated until this condition no longer holds.

  commands: Section, that will be executed at the end of each iteration of the loop. As well as *inits* also this section can hold multiple instructions, separated by comma. Usually this is used to increment or decrement the control-variables, defined in *inits*.

  As explained in the description, these loops are often used, because they give the possibility to the programmer, to execute statements at a certain point of the loop-execution. In the following example this behavior is used to initialize the variable *bottles* and to call the function *start()* at the beginning of the loop. After that, each iteration will print the current line-number n and the amount of bottles, which both are decreased/increased after each iteration of the loop.

```
var bottles;
var n=0;
var text = " bottles of beer on the wall.<br>";
//define function that prints start
var start = function(){
    document.write("Start:<br>")
}
// repeat loop until there are no more bottles
for ( bottles=99, start(); bottles>=0; bottles--, n++ ) {
    // print line-number and bottles
    document.write(n + ": " + bottles+text);
}
```

- **for-in-loop**

  Despite the same name, the *for-in*-loop is fundamentally different from the *for*-loop, what can also be seen immediately from the structure of the loop. Basically this loop is used to iterate over the property-names of an object:

  ```
  for ( var propName in object ) {
      //code here
  }
  ```

  propName: Variable which is used to store the property-name of the object

  object: Object, by which the property-names should be "explored"

  Literally this loop can be interpreted as "for each *property* in the *object*, do something". Because of this property, it is also convenient to use this function to iterate over all objects of an array (will be discussed later). Nevertheless it can also be used to "explore" all properties of an unknown object:

  ```
  // define object with two properties
  var obj=new Object();
  obj.Hello = "_";
  obj.beer = "!";
  // loop will print "Hello_beer!"
  for ( var name in obj ) {
      document.write(name+obj[name]);
  }
  ```

6 Arrays

Arrays in JavaScript can be created using the *Array()* constructor-function [2]. The "slots" of the arrays can then be accessed using the array-notation and the index, which always starts from zero. To determine the length of an array, the *length* property can be used (length of the array is always bigger than the biggest index of all objects that are held by the array). To be more efficient,

JavaScript implements arrays as "sparse arrays", that basically only need as much memory as the objects, which are held by the array. Therefore the real size of an array of the length 10, containing only two objects, could be the size of these two objects.

Due to the fact, that arrays in JavaScript can be seen as normal objects, they can be treated as associative arrays too. The data, that should be added to the array, can be also added as a normal property of the array-object. To iterate over all the objects that are held by the "associative" array, the *for-in*-loop can be used. An important side-effect is, that the data that was assigned to the array, using the "associative" way, is not covered by the *length*-property of the array.

7 What is JavaScript used for?

The main area, where JavaScript is used today is the internet. Nearly each website contains some piece of JavaScript-code. This code mostly is responsible for client-side manipulation of the website, but it can also be used to verify form inputs. Therefore it is possible to avoid unnecessary traffic between the client and the web server, because wrong inputs of the user can be detected soon by the client. The user may then be prompted to check the input before sending the whole data to the server. This opens up an other application field: pop-up windows. Pop-up windows are used to inform the user about events and can be realized easily using JavaScript. This property, for some users also represents a negative side of the language, because it is often abused to "bomb" the user with advertising. Nevertheless, JavaScript in most cases is used, as said before, for the client-side manipulation and dynamic construction of websites. In this context, the following concepts play an important role:

- **DOM:** The main idea behind the **D**ocument **O**bject **M**odel concept is, that each HTML-page can be represented as an object with certain sub-objects and properties, where all of these properties can be access and manipulated using JavaScript [5]. This is helpful, if the content of the website has to be changed dynamically, without reloading the whole page. For example the object `document.title` can be used to change the websites title. Due to this concept, it is possible to change the whole look or content of a website at the client-side without sending any request to a web server. Many animations are implemented using JavaScript too. A famous example for this area of application are animated timers, that often are embedded in websites to make the user wait for something.

- **AJaX: A**synchronous **J**avaScript **a**nd **X**ML describes the concept of sending requests to a web server in the background, while the actual page is displayed by the browser. This offers the possibility to load and change small pieces of a website without reloading it every time. Hence, this technology is used to build web applications, that asynchronously can communicate with a server. It is also possible to optimize the loading of websites using this concept, so that not the whole data is loaded at the

beginning but on demand, when it is really needed. The term XML is included in the name of AJaX, because the interchange of data mostly was realized using XML.

In combination, these two concepts are a powerful tool to realize web applications, that can be small tools on websites, like color-choosers, calculators, ..., or big applications like the *Google drive* [3] project.

7.1 How to embed JavaScript

JavaScript code can be embedded in HTML-pages in two different ways:

- **As HTML-tag:** The JavaScript code is deposited in the HTML-file that is displayed by the browser. To do this, the HTML-tag *script* can be used, where the property *type* has to be set to `text/javascript`.

  ```
  <script type="text/javascript">
       //some JavaScript-code here
  </script>
  ```

- **As external file:** The second possibility is to define the JavaScript-code in an external file and "include" it in the HTML-file. In the following example the code is defined in the file "scriptName.js", that is embedded using this instruction:

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

In both cases the defined code can be called from within the HTML-file. In the end, such a file should look like the following example, where a simple HTML-page is generated on which a button is displayed that should change the content if it is clicked:

```
<html>
<head>
<title>JavaScript</title>
</head>
<body>
<script type="text/javascript">
  // changing the buttons text
  function printHello(param){
    document.getElementById('bt1').value="Hello "+param;
  }
</script>
<!--button that calls printHello if its clicked-->
<input type="button" name="bt1"
value="Click me" id="bt1"
onclick="printHello('World');">
</body>
</html>
```

[3] More informations can be found here

8 Conclusion

JavaScript is a very common and popular scripting language, which today is used very often. The language owes its success to the fact, that it is a simple but powerful language, which can be very attractive for a huge crowd of developers, because of its property of a multi-paradigm language.

For the question, what would happen if JavaScript would just disappear over night from the scene, there exists a simple answer: Disable JavaScript in your web browser and see for yourself what happens. While for some websites you would not notice any difference, other sites could be limited in their functionality. Some others could have some malfunctions, or in the worst case the complete website is ruined and no longer accessible. Since JavaScript implements the ECMA standard, it could be pretty easy to switch to another implementation of the standard, that could replace the language (e.g. JScript from Microsoft or ActionScript by Adobe). Nevertheless, one thing is unquestionable: the internet as we know it today would certainly not exist, if the language had never been invented, because over the time, JavaScript had become a basic and essential tool to design and manage websites.

Going forward, I think that also in the future JavaScript will be an important language for technical areas. This argument is for example supported by the fact, that the applications for the free desktop-environment *Gnome* [4] in the future should be implemented increasingly using JavaScript. Some of the current components like the *Gnome-Shell* already contain JavaScript code. For this reason I think that it is possible to access completely new areas of application, that still could increase the popularity of the language.

[4] For more informations visit `http://www.gnome.org/`

References

[1] D. Flanagan. *JavaScript - Das umfassende Referenzwerk.* O'Reilly Media, 2012.

[2] M. D. Network and individual contributors. *JavaScript Reference* `https://developer.mozilla.org/en-US/docs/JavaScript/Reference`. [Online; accessed 04-February-2012].

[3] A. Rauschmayer. *The Past, Present, and Future of JavaScript.* O'Reilly Media, 2012.

[4] R. Steyer. *JavaScript. Einstieg für Anspruchsvolle.* Addison-Wesley Verlag, 2006.

[5] C. Wenz. *JavaScript und AJAX.* Galileo Computing, 2006.