JavaScript

Contents

- 1 Contents
- 2 Introduction
  - 2.1 Relation to Java
  - 2.2 About this book
  - 2.3 Audience
- 3 First Program
  - 3.1 Exercises
    - 3.1.1 Exercise 1-1
    - 3.1.2 Exercise 1-2
- 4 The SCRIPT Tag
  - 4.1 The script element
    - 4.1.1 Scripting Language
  - 4.2 Inline JavaScript
    - 4.2.1 Inline HTML comment markers
    - 4.2.2 Inline XHTML JavaScript
  - 4.3 Linking to external scripts
  - 4.4 Location of script elements
  - 4.5 Reference
- 5 Bookmarklets
  - 5.1 JavaScript URI scheme
  - 5.2 Using multiple lines of code
  - 5.3 The javascript Protocol in Links
  - 5.4 Examples
- 6 Lexical Structure
  - 6.1 Case Sensitivity
  - 6.2 Whitespace
  - 6.3 Comments
  - 6.4 Semicolons
  - 6.5 Literals
  - 6.6 Identifiers
  - 6.7 References
- 7 Reserved Words
  - 7.1 Reserved JavaScript keywords
  - 7.2 Words reserved for JavaScript in the future
- 8 Variables and Types
  - 8.1 Variable declaration
    - 8.1.1 Naming variables
  - 8.2 Primitive Types
    - 8.2.1 Boolean Type
    - 8.2.2 Numeric Types
    - 8.2.3 String Types

        - 8.3 Complex Types
            - 8.3.1 Array Type
            - 8.3.2 Object Types
        - 8.4 Further Reading
- 9 Numbers
    - 9.1 Basic Use
    - 9.2 The Math Object
        - 9.2.1 Methods
            - 9.2.1.1 random()
            - 9.2.1.2 max(int1, int2)
            - 9.2.1.3 min(int1, int2)
            - 9.2.1.4 floor(float)
            - 9.2.1.5 ceil(float)
            - 9.2.1.6 round(float)
        - 9.2.2 Properties
    - 9.3 Further reading
- 10 Strings
    - 10.1 Basic Use
    - 10.2 Properties and methods of the String() object
        - 10.2.1 replace(text, newtext)
        - 10.2.2 concat(text)
        - 10.2.3 toUpperCase()
        - 10.2.4 toLowerCase()
        - 10.2.5 length
        - 10.2.6 substring(start[, end])
        - 10.2.7 slice(start[, end])
        - 10.2.8 substr(start[, number of characters])
    - 10.3 Further reading
- 11 Dates
    - 11.1 Properties and methods
    - 11.2 Further Reading
- 12 Arrays
    - 12.1 Basic use
        - 12.1.1 Exercise
    - 12.2 Nested arrays
    - 12.3 Properties and methods of the Array() object
        - 12.3.1 concat()
        - 12.3.2 join() and split()
        - 12.3.3 pop() and shift()
        - 12.3.4 push() and unshift()
    - 12.4 Further reading
- 13 Operators
    - 13.1 Arithmetic Operators

    - 13.2 Bitwise Operators
    - 13.3 Assignment operators
    - 13.4 Increment operators
        - 13.4.1 Pre and post-increment operators
    - 13.5 Comparison operators
    - 13.6 Logical operators
    - 13.7 Other operators
- 14 Control Structures
    - 14.1 if
    - 14.2 while
    - 14.3 do... while
    - 14.4 for
    - 14.5 switch
- 15 Functions and Objects
    - 15.1 Functions
        - 15.1.1 Functions with arguments
    - 15.2 Objects
    - 15.3 The Date Object
    - 15.4 Defining New Objects
    - 15.5 this keyword
    - 15.6 Exceptions
    - 15.7 Further reading
- 16 Event Handling
    - 16.1 Event Handlers
        - 16.1.1 Event Attributes
    - 16.2 Standard event handlers
    - 16.3 Event Handlers as HTML attributes
- 17 Regular Expressions
    - 17.1 Compatibility
    - 17.2 Matching
    - 17.3 Replacement
    - 17.4 Test
    - 17.5 Modifiers
    - 17.6 Operators
    - 17.7 Function call
    - 17.8 See also
    - 17.9 External links
- 18 Optimization
    - 18.1 JavaScript Optimization
        - 18.1.1 Optimization Techniques
    - 18.2 Common Mistakes and Misconceptions
        - 18.2.1 String concatenation
- 19 Debugging

    - 19.1 JavaScript Debuggers
        - 19.1.1 Firebug
        - 19.1.2 Venkman JavaScript Debugger
        - 19.1.3 Internet Explorer debugging
        - 19.1.4 Safari debugging
        - 19.1.5 JTF: Javascript Unit Testing Farm
        - 19.1.6 jsUnit
        - 19.1.7 built-in debugging tools
    - 19.2 Common Mistakes
    - 19.3 Debugging Methods
        - 19.3.1 Following Variables as a Script is Running
    - 19.4 Browser Bugs
    - 19.5 browser-dependent code
    - 19.6 For further reading
- 20 DHTML
    - 20.1 alert messages
    - 20.2 Javascript Button and Alert Message Example:
    - 20.3 Javascript if() - else Example
    - 20.4 Two Scripts
    - 20.5 Simple Calculator
- 21 Finding Elements
    - 21.1 Simple Use
    - 21.2 Use of getElementsByTagName
- 22 Adding Elements
    - 22.1 Basic Usage
    - 22.2 Further Use
- 23 Changing Elements
- 24 Removing Elements
- 25 Code Structuring
- 26 Links
    - 26.1 Links
- 27 Useful Software Tools
    - 27.1 Editors / IDEs
    - 27.2 Engines and other tools

JavaScript

Contents

1. Welcome
    1. Introduction
    2. First Program
2. Basics
    1. Placing the Code
        1. The script element
        2. Bookmarklets
    2. Lexical Structure
        1. Reserved Words
    3. Variables and Types
        1. Numbers ● Strings ● Dates ● Arrays
    4. Operators
    5. Control Structures
    6. Functions and Objects
    7. Event Handling
    8. Program Flow
    9. Regular Expressions

Introduction

JavaScript is an interpreted computer programming language formalized in the ECMAScript language standard. JavaScript engines interpret and execute JavaScript. JavaScript engines may be designed for use as standalone interpretors, embedding in applications, or both. The first JavaScript engine was created by Netscape for embedding in their web browser. V8 is a JavaScript engine created for use in Google Chrome and may also be used as a standalone interpretor. Adobe Flash uses a JavaScript engine called ActionScript for development of Flash programs.

Relation to Java

JavaScript has no relation to Java aside from having a C-like syntax. Netscape developed JavaScript, and Sun Microsystems developed Java. The rest of this section assumes a background in programming, you may skip to the next section, if you like.

Variables have a static type (integer or string for example) which remains the same during the lifespan of a running program in Java, and have a dynamic type (Number or String for example) which can change during the lifespan of a running program in JavaScript. Variables must be declared prior to use in Java, and have a undefined value when referred to prior to assignment in JavaScript.

Java has an extensive collection of libraries which can be imported for use in programs. JavaScript does not provide any means to import libraries or external JavaScript code. JavaScript engines must extend the JavaScript language beyond the ECMAScript language standard, if additional functionality is desired, such as the required functionality provided by V8, or the Document Object Model found in many web browsers.

Java includes classes and object instances, and JavaScript uses prototypes.

About this book

This book is written as a tutorial, in the sense that all key concepts are explained. As such, it also contains exercises that are clearly marked as such at the end of a page or chapter. Answers for these exercises are also included.

The book can also be used as a reference. For this purpose, all keywords are mentioned and described.

Audience

This book assumes you have good knowledge and some experience in the use of computers, web browsers, text editors, and software development environments. You will not learn about HTML, CSS, Java, or website design in this book. Please consult an appropriate book to learn about these things.

First Program

Here is a single JavaScript statement, which creates a pop-up dialog saying "Hello World!":

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

For the browser to execute the statement, it must be placed inside a <script> element. This element describes which section of the HTML code contains executable code, and will be described in further detail later.

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

The <script> element should then be nested inside the <head> element of an HTML document. Assuming the page is viewed in a browser that has JavaScript enabled, the browser will execute (carry out) the statement as the page is loading.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Some Page</title>
    <script type="text/javascript">
      alert("Hello World!");
    </script>
  </head>
  <body>
    <p>The content of the web page.</p>
  </body>
</html>
```

This basic hello world program can then be used as a starting point for any new programs that you need to create.

Exercises

Exercise 1-1

Copy and paste the basic program in a file, save it on your hard disk as "exercise 1-1.html". You can run it in two ways:

1. By going to the file with a file manager, and opening it using a web browser (e.g. in Windows Explorer it is done with a double click)
2. By starting your browser, and then opening the file from the menu. For Firefox, that would be: Choose File in the menu, then Open File, then select the file.

What happens?

Answer

A dialog appears with the text: Hello World!

Exercise 1-2

Save the file above as "exercise 1-2.html". Replace the double quotes in the line *alert("Hello World!");* with single quotes, so it reads *alert('Hello World!');* and save the result. If you open this file in the browser, what happens?

Answer

Nothing changes. A dialog appears with the text: Hello World! The double quotes and the single quotes are equivalent.

The SCRIPT Tag

The script element

All JavaScript, when placed in an HTML document, needs to be within a script element. A script element is used to link to an external JavaScript file, or to contain inline scripting (script snippets in the HTML file). A script element to link to an external JavaScript file looks like:

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

while a script element that contains inline JavaScript looks like:

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

Inline scripting has the advantage that both your HTML and your JavaScript are in one file, which is convenient for quick development and testing. Having your JavaScript in a separate file is recommended for JavaScript functions which can potentially be used in more than one page, and also to separate content from behaviour.

Scripting Language

The script element will work in most browsers, because JavaScript is currently the default scripting language. It is strongly recommended though to specify what type of script you are using in case the default scripting language changes.

The scripting language can be specified individually in the script element itself, and you may also use a meta tag in the head of the document to specify a default scripting language for the entire page.

```
<meta http-equiv="Content-Script-Type" content="text/javascript" />
```

While the text/javascript was formally obsoleted in April 2006 by RFC 4329 [1] in favour of application/javascript, it is still preferable to continue using text/javascript due to old HTML validators and old web browsers such as Internet Explorer 5 which are unable to understand application/javascript. [2]

Inline JavaScript

Using inline JavaScript allows you to easily work with HTML and JavaScript within the same page. This is commonly used for temporarily testing out some ideas, and in situations where the script code is specific to that one page.

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

Inline HTML comment markers

The inline HTML comments are to prevent older browsers that do not understand the script element from displaying the script code in plain text.

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

Older browsers that do not understand the script element will interpret the entire content of the script element above as one single HTML comment, beginning with "<!--" and ending with "-->", effectively ignoring the script completely. If the HTML comment was not there, the entire script would be displayed in plain text to the user by these browsers.

Current browsers that know about the script element will ignore the *first* line of a script element if it starts with "<!--". In the above case, the first line of the actual JavaScript code is therefore the line "// JavaScript code here".

The last line of the script, "// -->", is a one line JavaScript comment which prevents the HTML end comment tag "-->" from being interpreted as JavaScript.

The use of comment markers is rarely required nowadays, as the browsers that do not recognise the script element are virtually non-existent. These early browsers were Mosaic, Netscape 1, and Internet Explorer 2. From Netscape 2.0 in December 1995 and Internet Explorer 3.0 in August 1996. Those browsers were able to interpret javascript.[3] Any modern browser that doesn't support JavaScript, recognizes the <script> tag and does not display it to the user.

Inline XHTML JavaScript

In XHTML, the method is somewhat different:

```
<script type="text/javascript">
   // <![CDATA[
   // JavaScript code here
   // ]]>
</script>
```

Note that both the <![CDATA[tags are commented out. The // prevents the browser from mistakenly interpreting the <![CDATA[as a Javascript statement (which would be a syntax error).

Linking to external scripts

JavaScript is commonly stored in a file so that it may be used by many web pages on your site. This makes it much easier for updates to occur and saves space on your server. This method is recommended for separating behavior (JavaScript) from content ((X)HTML) and it prevents the issue of incompatibility with inline comments in XHTML and HTML.

Add src="*script.js*" to the opening script tag. Replace *script.js* with the path to the .js file containing the JavaScript.

Because the server provides the content type when the file is requested, specifying the type is optional when linking to external scripts. It's still advised to specify the type as text/javascript, in case the server isn't set up correctly, and to prevent HTML validation complaints.

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

Location of script elements

The script element may appear almost anywhere within the HTML file.

A standard location is within the head element. Placement within the body however is allowed.

```
<!DOCTYPE html>
<html>
<head>
   <title>Web page title</title>
   <script type="text/javascript" src="script.js"></script>
</head>
<body>
<!-- HTML code here -->
</body>
</html>
```

There are however some best practices for speeding up your web site [4] from the Yahoo! Developer Network that specify a different placement for scripts, to put scripts at the bottom, just before the </body> tag. This speeds up downloading, and also allows for direct manipulation of the DOM while the page is loading.

```
<!DOCTYPE html>
<html>
<head>
   <title>Web page title</title>
</head>
<body>
<!-- HTML code here -->
<script type="text/javascript" src="script.js"></script>
</body>
</html>
```

Bookmarklets

Bookmarklets are one line scripts stored in the URL field of a bookmark. Bookmarklets have been around for a long time so they will work in older browsers.

JavaScript URI scheme

You should be familiar with URL that start with schemes like http and ftp, e.g.

http://en.wikibooks.org/. There is also the javascript scheme, which is used to start every bookmarklet.

javascript:alert("Hello, world!");

Using multiple lines of code

Since you cannot have line breaks in bookmarklets you must use a semicolon at the end of each code statement instead.

javascript:name=prompt("What is your name?");alert("Hello, "+name);

The javascript Protocol in Links

The javascript protocol can be used in links. This may be considered bad practice because it prevents access for or confuses users who have disabled JavaScript. See Best Practices.

blue background

Examples

A large quantity of links may be found on bookmarklets.com, which show a variety of features that can be performed within Javascript.

Lexical Structure

Case Sensitivity

JavaScript is case sensitive. This means that Hello() is not the same as HELLO() or hello()

Whitespace

Whitespace can be; extra indents, line breaks, and spaces. Javascript ignores it, but it makes the code easier for people to read.

The following is JavaScript with very little whitespace.

```
function filterEmailKeys(evt){
evt=evt||window.event;
var charCode=evt.charCode||evt.keyCode;
var char=String.fromCharCode(charCode);
if(/[a-zA-Z0-9_\-\.@]/.exec(char))
return true;
return false;
}
```

The following is the same JavaScript with a typical amount of whitespace.

```
function filterEmailKeys(evt) {
    evt = evt || window.event;
    var charCode = evt.charCode || evt.keyCode;
    var char = String.fromCharCode(charCode);
    if (/[a-zA-Z0-9_\-\.@]/.exec(char)) {
        return true;
    }
    return false;
}
```

The following is the same JavaScript with a lot of whitespace.

```
function filterEmailKeys( evt )
  {
    evt = evt || window.event;

    var charCode = evt.charCode || evt.keyCode;
    var char = String.fromCharCode ( charCode );

    if ( /[a-zA-Z0-9_\-\.@]/.exec ( char ) )
      {
        return true;
      }

    return false;
  }
```

Comments

Comments allow you to leave notes in your code to help other people understand it. They also allow you to comment out code that you want to hide from the parser, but you don't want to delete.

Single-line comments

A double slash, //, turns all of the following text on the same line into a comment that will not be processed by the JavaScript interpreter.

```
// Shows a welcome message
alert("Hello, World!")
```

Multi-line comments

Multi-line comments are begun with slash asterisk, /*, and end with the reverse asterisk slash, */.

Here is an example of how to use the different types of commenting techniques.

```
/* This is a multi-line comment
that contains multiple lines
of commented text. */
var a = 1;
/* commented out to perform further testing
a = a + 2;
a = a / (a - 3); // is something wrong here?
*/
alert('a: ' + a);
```

Semicolons

In many computer languages semicolons are required at the end of each code statement. In JavaScript the use of semicolons is optional, as a new line indicates the end of the statement. This is automatic semicolon insertion and the rules for it are quite complex [1]. Leaving out semicolons and allowing the parser to automatically insert them can create complex problems.

```
a = b + c
(d + e).print()
```

The above code is not interpreted as two statements. Because of the parentheses on the second line, JavaScript interprets the above as if it were

```
a = b + c(d + e).print();
```

when instead you may have meant it to be interpreted as

```
a = b + c;
(d + e).print();
```

Even though semicolons are optional, it's preferable to end statements with a semicolon to prevent any misunderstandings from taking place.

Literals

A literal is a hard coded value. Literals provide a means of expressing specific values in your script. For example, at the right of equal:

```
var myLiteral = "a fixed value";
```

There are several types of literals available. The most common are the string literals, but there are also integer and floating-point literals, array and boolean literals, and object literals.

Example of an object literal:

```
var myObject = { name:"value", anotherName:"anotherValue"};
```

Details of these different types are covered in Variables and Types.

Identifiers

An identifier is a name for a piece of data such as a variable, array, or function. There are rules:

- Letters, dollar signs, underscores, and numbers are allowed in identifiers.
- The first character cannot be a number.

Examples of valid identifiers:

- u
- $hello
- _Hello
- hello90

References

1. Standard ECMA-262 ECMAScript Language Specification, Chapter 7.9 - Automatic Semicolon Insertion

Reserved Words

This page contains a list of reserved words in JavaScript, which cannot be used as names of variables, functions or other objects.

Reserved JavaScript keywords

Current list as of Version 5.1 [1]

- break
- case
- catch
- continue
- debugger
- default
- delete
- do
- else
- ~~false~~
- finally
- for
- function
- if
- in
- instanceof
- new
- ~~null~~
- return
- switch
- this
- throw
- ~~true~~
- try
- typeof
- var
- void
- while
- with

Words reserved for JavaScript in the future

Current list as of Version 5.1 [2]

- ~~abstract~~

JavaScript

- ~~boolean~~
- ~~byte~~
- ~~char~~
- class
- const
- ~~double~~
- enum
- export
- extends
- ~~final~~
- ~~float~~
- ~~goto~~
- implements
- import
- ~~int~~
- interface
- let
- ~~long~~
- ~~native~~
- package
- private
- protected
- public
- ~~short~~
- static
- super
- ~~synchronized~~
- ~~throws~~
- ~~transient~~
- ~~volatile~~

Variables and Types

JavaScript is a loosely typed language. This means that you can use the same variable for different types of information, but you may also have to check what type a variable is yourself if the differences matter. For example, if you wanted to add two numbers, but one variable turned out to be a string, the result wouldn't necessarily be what you expected.

Variable declaration

Variables are commonly explicitly declared by the var statement, as shown below:

```
var c;
```

The above variable is created, but has the default value of undefined. To be of value, the variable needs to be initialized:

```
var c = 0;
```

After being declared, a variable may be assigned a new value which will replace the old one:

```
c = 1;
```

But make sure to declare a variable with var before (or while) assigning to it; otherwise you will create a "scope bug."

Naming variables

When naming variables there are some rules that must be obeyed:

- Upper case and lower case letters of the alphabet, underscores, and dollar signs can be used
- Numbers are allowed after the first character
- No other characters are allowed
- Variable names are case sensitive: different case implies a different name
- A variable may not be a reserved word

Primitive Types

Primitive types are types provided by the system, in this case by javascript. Primitive type for javascript are booleans, numbers and text. In addition to the primitive types, users may define their own classes.

The primitive types are treated by Javascript as value types and when you pass them around they go as values. Some types, such as string, allow method calls.

Boolean Type

Boolean variables can only have 2 possible values, true or false.

```
var mayday = false;
var birthday = true;
```

Numeric Types

You can use Integer and Double types on your variables, but they are treated as a numeric type.

```
var sal = 20;
var pal = 12.1;
```

In ECMA Javascript your number literals can go from 0 to -+1.79769e+308. And because 5e-324 is the smallest infinitesimal you can get, anything smaller is rounded to 0.

String Types

The String and char types are all strings, so you can build any string literal that you wished for.

```
var myName = "Some Name";
var myChar = 'f';
```

Complex Types

A complex type is an object, be it either standard or custom made. Its home is the heap and goes everywhere by reference.

Array Type

In Javascript, all Arrays are untyped, so you can put everything you want in an Array and worry about that later. Arrays are objects, they have methods and properties you can invoke at will. For example, the .length property indicates how many items are currently in the array. If you add more items to the array, the value of the .length gets larger. You can build yourself an array by using the statement new followed by Array, as shown below.

```
var myArray = new Array(0, 2, 4);
var myOtherArray = new Array();
```

Arrays can also be created with the array notation, which uses square brackets:

```
var myArray = [0, 2, 4];
var myOtherArray = [];
```

Arrays are accessed using the square brackets:

```
myArray[2] = "Hello";
var text = myArray[2];
```

There is no limit to the number of items that can be stored in an array.

Object Types

An object within Javascript is created using the new operator:

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

Objects can also be created with the object notation, which uses curly braces:

```
var myObject = {};
```

JavaScript Objects can be built using inheritance and overriding, and you can use polymorphism. There are no scope modifiers, with all properties and methods having public access. More information on creating objects can be found in Object Oriented Programming.

You can access browser built-in objects and objects provided through browser JavaScript extensions.

Further Reading

"Values, variables, and literals". MDN. May 28, 2013. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Values,_variables,_and_literals. Retrieved 20/06/2013.

Numbers

JavaScript implement numbers as floating point values, that is, they're attaining decimal values as well as whole number values.

Basic Use

To make a new number, a simple initialization suffices:

```
var foo = 0; // or whatever number you want
```

After you have made your number, you can then modify it as necessary. Numbers can be modified or assigned using the operators defined within JavaScript.

```
foo = 1; //foo = 1
foo += 2; //foo = 3 (the two gets added on)
foo -= 2; //foo = 1 (the two gets removed)
```

Number literals define the number value. In particular:

- They appear as a set of digits of varying length.
- Negative literal numbers have a minus sign before the set of digits.
- Floating point literal numbers contain one decimal point, and may optionally use the E notation with the character e.
- An integer literal may be prepended with "0", to indicate that a number is in base-8. (8 and 9 are not octal digits, and if found, cause the integer to be read in the normal base-10).
- An integer literal may also be found with prefixed "0x", to indicate a hexadecimal number.

The Math Object

Unlike strings, arrays, and dates, the numbers aren't objects, so they don't contain any methods that can be accessed by the normal dot notation. Instead a certain Math object provides usual numeric functions and constants as methods and properties. The methods and properties of the Math object are referenced using the *dot operator* in the usual way, for example:

```
var varOne = Math.ceil(8.5);
var varPi = Math.PI;
var sqrt3 = Math.sqrt(3);
```

Methods

random()

Generates a pseudo-random number.

```
var myInt = Math.random();
```

max(int1, int2)

Returns the highest number from the two numbers passed as arguments.

```
var myInt = Math.max(8, 9);
document.write(myInt); //9
```

min(int1, int2)

Returns the lowest number from the two numbers passed as arguments.

```
var myInt = Math.min(8, 9);
document.write(myInt); //8
```

floor(float)

Returns the greatest integer less than the number passed as an argument.

```
var myInt = Math.floor(90.8);
document.write(myInt); //90;
```

ceil(float)

Returns the least integer greater than the number passed as an argument.

```
var myInt = Math.ceil(90.8);
document.write(myInt); //91;
```

round(float)

Returns the closest integer to the number passed as an argument.

```
var myInt = Math.round(90.8);
document.write(myInt); //91;
```

Properties

Properties of the Math Object are most commonly used constants.

- **PI:** Returns the value of pi.
- **E:** Returns the constant e.
- **SQRT2:** Returns the square root of 2.
- **LN10:** Returns the natural logarithm of 10.
- **LN2:** Returns the natural logarithm of 2.

Strings

A **string** is a type of variable which stores a string (chain of characters).

Basic Use

To make a new string, you can make a variable and give it a value of new String().

```
var foo = new String();
```

But, most developers skip that part and use a string literal:

```
var foo = "my string";
```

After you have made your string, you can edit it as you like:

```
foo = "bar"; //foo = "bar"
foo = "barblah"; //foo = "barblah"
foo += "bar"; //foo = "barblahbar"
```

A string literal is normally delimited by the ′ or ″ character, and can normally contain almost any character. Common convention differs on whether to use single quotes or double quotes for strings. Some developers are for single quotes (Crockford, Amaram, Sakalos, Michaux) while others are for double quotes (NextApp, Murray, Dojo). Whichever method you choose, try to be consistent in how you apply it.

Due to the delimiters, it's not possible to directly place either the single or double quote within the string when it's used to start or end the string. In order to work around that limitation, you can either switch to the other type of delimiter for that case, or place a backslash before the quote to ensure that it appears within the string:

```
foo = 'The cat says, "Meow!"';
foo = "The cat says, \"Meow!\"";
foo = "It's \"cold\" today.";
foo = 'It\'s "cold" today.';
```

Properties and methods of the String() object

As with all objects, Strings have some methods and properties.

replace(text, newtext)

The replace() function returns a string with content replaced. Only the first occurrence is replaced.

```
var foo = "foo bar foo bar foo";
var newString = foo.replace("bar", "NEW!")
alert(foo); //foo bar foo bar foo
alert(newString); //foo NEW! foo bar foo
```

As you can see, the replace() function only returns the new content and does not modify the 'foo' object.

concat(text)

The concat() function joins two strings.

```
var foo = "Hello";
var bar = foo.concat(" World!")
alert(bar); //Hello World!
```

toUpperCase()

This function returns the current string in upper case.

```
var foo = "Hello!";
alert(foo.toUpperCase()); // HELLO!
```

toLowerCase()

This function returns the current string in lower case.

```
var foo = "Hello!";
alert(foo.toLowerCase()); // hello!
```

length

Returns the length as an integer.

```
var foo = "Hello!";
alert(foo.length); // 6
```

substring(start[, end])

Substring extracts characters from the *start* position

```
"hello".substring(1); // "ello"
```

When the *end* is provided, they are extracted up to but not including the end position.

```
"hello".substring(1, 3); // "el"
```

Substring always works from left to right. If the *start* position is larger than the *end* position, Substring will swap the values; although sometimes useful, this is not always what you want; different behavior is provided by slice.

```
"hello".substring(3, 1); // "el"
```

slice(start[, end])

Slice extracts characters from the *start* position, essentially the same as substring

```
"hello".slice(1); // "ello"
```

When the *end* is provided, they are extracted up to but not including the end position.

```
"hello".slice(1, 3); // "el"
```

Slice allows you to extract text referenced from the end of the string by using negative indexing.

```
"hello".slice(-4, -2); // "el"
```

Unlike Substring, the Slice method never swaps the *start* and *end* positions. If the *start* is after the *end*, Slice will attempt to extract the content as presented, but will most likely provide unexpected results.

```
"hello".slice(3, 1); // ""
```

substr(start[, number of characters])

substr extracts characters from the *start* position, essentially the same as substring/slice

```
"hello".substr(1); // "ello"
```

When the *number of characters* is provided, they are extracted by count.

```
"hello".substr(1, 3); // "ell"
```

Dates

A Date is an object that contains a given time to millisecond precision.

Unlike strings and numbers, the date must be explicitly created with the new operator.

```
var date = new Date(); // Create a new Date object with the current date and time.
```

The Date object may also be created using parameters passed to its constructor. By default, the Date object contains the current date and time found on the computer, but can be set to any date or time desired.

```
var time_before_2000 = new Date(1999, 11, 31, 23,59,59,999);
```

The date can also be returned as an integer. This can apply to seeding a PRNG method, for example.

```
var integer_date = +new Date; // Returns a number, like 1362449477663.
```

The date object normally stores the value within the local time zone. If UTC is needed, there are a set of functions available for that use.

The Date object does not support non-CE epochs, but can still represent almost any available time within its available range.

Properties and methods

Properties and methods of the Date() object:

setFullYear(year), getFullYear()
: Stores or retrieves the full 4-digit year within the Date object.

setMonth(month, day)
: Sets the month within the Date object, and optionally the day within the month. [0 - 11]. The Date object uses 0 as January instead of 1.

getMonth()
: Returns the current month. [0 - 11]

getDate()
: Returns the day of the month. [0 - 30]

getDay()
: Returns the day of the week within the object. [0 - 6]. Sunday is 0, with the other days of the week taking the next value.

parse(text)
: Reads the string *text*, and returns the number of milliseconds since January 1, 1970.

Arrays

An **array** is a type of variable that stores a collection of variables. Arrays in JavaScript are zero-

based - they start from zero. (instead of foo[1], foo[2], foo[3], JavaScript uses foo[0], foo[1], foo[2].)

Basic use

To make a new array, make a variable and give it a value of new Array().

```
var foo = new Array()
```

After defining it, you can add elements to the array by using the variable's name, and the name of the array element in square brackets.

```
foo[0] = "foo";
foo[1] = "fool";
foo[2] = "food";
```

You can call an element in an array the same way.

```
alert(foo[2]);
//outputs "food"
```

You can define and set the values for an array with shorthand notation.

```
var foo = ["foo", "fool", "food"];
```

Exercise

Make an array with "zzz" as one of the elements, and then make an alert box using that element.

Nested arrays

You can put an array in an array.

The first step is to simply make an array. Then make an element (or more) of it an array.

```
var foo2 = new Array();
foo2[0] = new Array();
foo2[1] = new Array();
```

To call/define elements in a nested array, use two sets of square brackets.

```
foo2[0][0] = "something goes here";
foo2[0][1] = "something else";
foo2[1][0] = "another element";
foo2[1][1] = "yet another";
alert(foo2[0][0]); //outputs "something goes here"
```

You can use shorthand notation with nested arrays, too.

```
var foo2 = [ ["something goes here", "something else"], ["another element", "yet another"] ];
```

So that they're easier to read, you can spread these shorthand notations across multiple lines.

```
var foo2 = [
   ["something goes here", "something else"],
   ["another element", "yet another"]
];
```

Properties and methods of the Array() object

concat()

The concat() method returns the combination of two or more arrays.

To use it, first you need two or more arrays to combine.

```
var arr1 = ["a","b","c"];
var arr2 = ["d","e","f"];
```

Then, make a third array and set its value to arr1.concat(arr2).

```
var arr3 = arr1.concat(arr2) //arr3 now is: ["a","b","c","d","e","f"]
```

join() and split()

The join() method combines all the elements of an array into a single string, separated by a specified delimiter. If the delimiter is not specified, it is set to a comma. The split() is the opposite and splits up the contents of a string as elements of an array, based on a specified delimiter.

To use join(), first make an array.

```
var abc = ["a","b","c"];
```

Then, make a new variable and set it to abc.join().

```
var a = abc.join(); //"a,b,c"
```

You can also set a delimiter.

```
var b = abc.join("; "); //"a; b; c"
```

To convert it back into an array with the String object's split() method.

```
var a2 = a.split(","); //["a","b","c"]
var b2 = b.split("; "); //["a","b","c"]
```

pop() and shift()

The pop() method removes and returns the last element of an array. The shift() method does the same with the first element. (note: The shift() method also changes all the index numbers of the

array. For example, arr[0] is removed, arr[1] becomes arr[0], arr[2] becomes arr[1], and so on.)

First, make an array.

```
var arr = ["0","1","2","3"];
```

Then use pop() or shift().

```
alert(arr); //outputs "0,1,2,3"
alert(arr.pop()); //outputs "3"
alert(arr); //outputs "0,1,2"
alert(arr.shift()); //outputs "0"
alert(arr); //outputs "1,2"
```

push() and unshift()

The push() and unshift() methods reverse the effect of pop() and shift(). The push() method adds an element to the end of an array and returns its new length. The unshift() method does the same with the beginning of the array (and like shift(), also adjusts the indexes of the elements.)

```
arr.unshift("0"); //"0,1,2"
arr.push("3"); //"0,1,2,3"
```

Operators

Arithmetic Operators

Javascript has the arithmetic operators +, -, *, /, and %. These operators function as the addition, subtraction, multiplication, division, and modulus operators, and operate very similarly to other languages.

```
var a = 12 + 5; // 17
var b = 12 - 5; // 7
var c = 12 * 5; // 60
var d = 12 / 5; // 2.4 - division results in floating point numbers.
var e = 12 % 5; // 2 - the remainder of 12/5 in integer math is 2.
```

Some mathematical operations, such as dividing by zero, cause the returned variable to be one of the error values - for example, infinity, or NaN.

The return value of the modulus operator maintains the sign of the first operand.

The + and - operators also have unary versions, where they operate only on one variable. When used in this fashion, + returns the number representation of the object, while - returns its negative counterpart.

```
var a = "1";
var b = a;  // b = "1": a string
var c = +a;  // c = 1: a number
var d = -a;  // d = -1: a number
```

+ is also used as the string concatenation operator: If any of its arguments is a string or is otherwise *not* a number, any non-string arguments are converted to strings, and the 2 strings are concatenated. For example, 5 + [1,2,3] evaluates to the string "51,2,3". More usefully, str1 + " " + str2 returns str1 concatenated with str2, with a space between.

All other arithmetic operators will attempt to convert their arguments into numbers before evaluating. Note that unlike C or Java, the numbers and their operation results are not guaranteed to be integers.

Bitwise Operators

There are 7 bitwise operators, &, |, ^, ~, >>, <<, and >>>.

These operators convert their operands to integers (truncating any floating point towards 0), and perform the specified bitwise operation on them. The logical bitwise operators, &, |, and ^, perform the *and*, *or*, and *xor* on each individual bit and provides the return value. The ~ (*not* operator) inverts all bits within an integer, and usually appears in combination with the logical bitwise operators.

Two bit shift operators, >>, <<, move the bits in one direction which has a similar effect to multiplying or dividing by a power of two. The final bit-shift operator, >>>, operates the same

way, but does not preserve the sign bit when shifting.

These operators are kept for parity with the related programming languages but are unlikely to be used in most JavaScript programs.

Assignment operators

The assignment operator = assigns a value to a variable. Primitive types, such as strings and numbers are assigned directly, however function and object names are just pointers to the respective function or object. In this case, the assignment operator only changes the reference to the object rather than the object itself. For example, after the following code is executed, "0,1,0" will be alerted, even though setA was passed to the alert, and setB was changed. This is because they are two references to the same object.

```
setA = [ 0, 1, 2 ];
setB = setA;
setB[2] = 0;
alert(setA);
```

Similarly, after the next bit of code is executed, x is a pointer to an empty array.

```
z = [5];
x = z;
z.pop();
```

All the above operators have corresponding assignment operators of the form *operator*=. For all of them, x *operator*= y is just a convenient abbreviation for x = x *operator* y.

- +=
- -=
- *=
- /=
- %=
- &=
- |=
- ^=
- >>=
- <<=
- >>>=

For example, a common usage for += is in a for loop

```
var els = document.getElementsByTagName('h2');
var i;
for (i = 0; i < els.length; i += 1) {
  // do something with els[i]
}
```

Increment operators

There are also the increment and decrement operators, ++ and --. a++ increments a and returns the old value of a. ++a increments a and returns the new value of a. The decrement operator functions similarly, but reduces the variable instead.

As an example, the last three lines all perform the same task:

```
var a = 1;
a = a + 1;
a += 1;
++a;
```

Pre and post-increment operators

Increment operators may be applied before or after a variable. When they are applied before a variable they are pre-increment operators, and when they are applied after a variable they are post-increment operators.

The choice of which to use changes how they affect operations.

```
// increment occurs before a is assigned to b
var a = 1;
var b = ++a; // a = 2, b = 2;

// increment occurs to c after c is assigned to d
var c = 1;
var d = c++; // c = 2, d = 1;
```

Due to the possibly confusing nature of pre and post-increment behaviour, code can be easier to read if the increment operators are avoided.

```
// increment occurs before a is assigned to b
var a = 1;
a += 1;
var b = a; // a = 2, b = 2;

// increment occurs to c after c is assigned to d
var c = 1;
var d = c;
c += 1; // c = 2, d = 1;
```

Comparison operators

The comparison operators determine if the two operands meet the given condition.

- == - returns true if the two operands are equal. This operator may ignore an operand's type (e.g. a string as an integer).
- === - returns true if the two operands are identical. This operator will not ignore the operands' types, and only returns true if they are the same type and value.
- != - returns true if the two operands are not equal. This operator may ignore an operand's

type (e.g. a string as an integer).
- !== - returns true if the two operands are not identical. This operator will not ignore the operands' types, and only returns false if they are the same type and value.
- > - Returns true if the first operand is greater than the second one.
- >= - Returns true if the first operand is greater than or equal to the second one.
- < - Returns true if the first operand is less than the second one.
- <= - Returns true if the first operand is less than or equal to the second one.

Be careful when using == and != as the may ignore one of the terms type being compared. This can lead to strange and non-intuitive situations, such as:

```
0 == '' // true
0 == '0' // true
false == 'false' // false; (''Boolean to string'')
false == '0' // true (''Boolean to string'')
false == undefined // false
false == null // false (''Boolean to null'')
null == undefined // true
```

For stricter compares use === and !==

```
0 === '' // false
0 === '0' // false
false === 'false' // false
false === '0' // false
false === undefined // false
false === null // false
null === undefined // false
```

Logical operators

- && - and
- || - or
- ! - not

The logical operators are *and*, *or*, and *not*. The && and || operators accept two operands and provides their associated logical result, while the third accepts one, and returns it's logical negation. && and || are short circuit operators. If the result is guaranteed after evaluation of the first operand, it skips evaluation of the second operand.

Technically, the exact return value of these two operators is also equal to the final operand that it evaluated. Due to this, the && operator is also known as the guard operator, and the || operator is also known as the default operator.

```
function handleEvent(evt) {
    evt = evt || window.event;
    var targ = evt.target || evt.srcElement;
    if (targ && targ.nodeType === 1 && targ.nodeName === 'A') {
        // ...
    }
}
```

The *!* operator determines the inverse of the given value, and returns the boolean: true values become false, or false values become true.

Note: Javascript represents false by either a boolean false, the number 0, an empty string, or the built in undefined or null type. Any other value is treated as true.

Other operators

- ? :

The ? : operator (also called the "ternary" operator).

```
var targ = (a == b) ? c : d;
```

Be cautious though in its use. Even though you can replace verbose and complex if/then/else chains with ternary operators, it may not be a good idea to do so. You can replace

```
if (p && q) {
    return a;
} else {
    if (r != s) {
        return b;
    } else {
        if  (t || !v) {
            return c;
        } else {
            return d;
        }
    }
}
```

with

```
return (p && q) ? a
  : (r != s) ? b
  : (t || !v) ? c
  : d
```

The above example is a poor coding style/practice. When other people edit or maintain your code (which could very possibly be you) it becomes much more difficult to understand and work with the code.

It is better to instead make the code more understandable. Some of the excessive conditional nesting can be removed from the above example.

```
if (p && q) {
    return a;
}
if (r != s) {
    return b;
}
if  (t || !v) {
    return c;
} else {
```

```
    return d;
}
```

- typeof x returns a string describing the type of x.
- o instanceof c tests whether o is an object created by the constructor c.
- delete x unbinds x.
- new cl creates a new object of type *cl*. The *cl* operand must be a constructor function.

Control Structures

The control structures within Javascript allow the program flow to change within a unit of code or function. These statements can determine whether or not given statements are executed, as well as repeated execution of a block of code.

With the exception of the *switch* control structure, all control structures can operate either on a statement or a block of code enclosed with braces (*{* and *}*). The same structures utilize booleans to determine whether or not a block gets executed, where any defined variable that is neither zero or an empty string is treated as true.

if

The *if* statement is straight forward - if the given expression is true, the statement or statements will be executed. Otherwise, they are skipped.

```
if (a === b) {
  document.body.innerHTML += "a equals b";
}
```

The *if* statement may also consist of multiple parts, incorporating *else* and *else if* sections. These keywords are part of the *if* statement, and identify the code blocks that are executed if the preceding condition is false.

```
if (a === b) {
  document.body.innerHTML += "a equals b";
} else if (a === c) {
  document.body.innerHTML += "a equals c";
} else {
  document.body.innerHTML += "a does not equal either b or c";
}
```

while

The *while* statement executes a given statement as long as a given expression is true. For example, the code block below will increase the variable *c* to 10:

```
while (c < 10) {
  c += 1;
  // ...
}
```

This control loop also recognizes the *break* and *continue* keywords. The *break* keyword causes the immediate termination of the loop, allowing for the loop to terminate from anywhere within the block.

The *continue* keyword finishes the current iteration of the *while* block or statement, and checks the condition to see if it is true. If it is true, the loop commences again.

do... while

The *do ... while* statement executes a given statement as long as a given expression is true - however, unlike the *while* statement, this control structure will always execute the statement or block at least once. For example, the code block below will increase the variable *c* to 10:

```
do {
  c += 1;
} while (c < 10)
```

As with *while*, *break* and *continue* are both recognized and operate in the same manner. *break* exits the loop, and *continue* checks the condition before attempting to restart the loop.

for

The *for* statement allows greater control over the condition of iteration. While it has a conditional statement, it also allows a pre-loop statement, and post-loop increment without affecting the condition. The initial expression is executed once, and the conditional is always checked at the beginning of each loop. At the end of the loop, the increment statement executes before the condition is checked once again. The syntax is:

for (<initial expression>;<condition>;<final expression>)

The *for* statement is usually used for integer counters:

```
var c;
for (c = 0; c < 10; c += 1) {
  // ...
}
```

While the increment statement is normally used to increase a variable by one per loop iteration, it can contain any statement, such as one that decreases the counter.

Break and *continue* are both recognized. The *continue* statement will still execute the increment statement before the condition is checked.

switch

The *switch* statement evaluates an expression, and determines flow control based on the result of the expression:

```
switch(i) {
case 1:
  // ...
  break;
case 2:
  // ...
  break;
default:
  // ...
  break;
}
```

When *i* gets evaluated, it's value is checked against each of the *case* labels. These *case* labels appear in the *switch* statement and, if the value for the case matches i, continues the execution at that point. If none of the case labels match, execution continues at the *default* label (or skips the switch statement entirely if none is present.)

Case labels may only have constants as part of their condition.

The *break* keyword exits the *switch* statement, and appears at the end of each case in order to prevent undesired code from executing. While the *break* keyword may be omitted (for example, you want a block of code executed for multiple cases), it may be considered bad practice doing so.

The *continue* keyword does not apply to *switch* statements.

Functions and Objects

Functions

A **function** is an action to take to complete a goal, objective, or task. Functions allow you to split a complex goal into simpler tasks, which make managing and maintaining scripts easier. A **parameter** or **argument** is data which is passed to a function to effect the action to be taken. Functions can be passed zero or more arguments. A function is executed when a call to that function is made anywhere within the script, the page, an external page, or by an event. Functions are always guaranteed to return some value when executed. The data passed to a function when executed is known as the function's input and the value returned from an executed function is known as the function's output.

Functions can be constructed in three main ways. We begin with three "Hello, World!" examples:

Way 1

```
function hello() {
  alert("Hello, World!");
}
```

Way 2

```
var hello = function() {
  alert("Hello, World!");
};
```

Way 3

```
var hello = new Function(
  'alert("Hello, World!");'
);
```

Each function:

- can be called with hello()
- does not expect any arguments
- performs an action to alert the user with a message
- undefined is returned when execution is finished

The hello function can be changed to allow you to say hello to someone specific through the use of arguments:

Way 1

```
function hello(who) {
  alert("Hello, " + who + "!");
}
```

Way 2

```
var hello = function(who) {
  alert("Hello, " + who + "!");
};
```

Way 3

```
var hello = new Function('who',
  'alert("Hello, " + who + "!");'
);
```

Each function:

- can be called with hello
- expects one argument to be passed
- performs an action to alert the user with a message
- undefined is returned when execution is finished

Each function can be called in several ways:

Way 1

```
hello("you");
```

Way 2

```
hello.call(window, "you");
```

Way 3

```
hello.apply(window, ["you"]);
```

Let's put this together on a sample web page. Call the function once when the page is loaded. Call the function again whenever a button is clicked.

```
<html>
 <head><title>Some Page</title></head>
 <body>
  <button id="msg">greeting</button>
  <script type="text/javascript">

   function hello() {
     alert("Hello World!");
   }

   document.getElementById("msg").onclick = hello;

   hello();

  </script>
 </body>
</html>
```

Functions with arguments

Let's start with a quick example, then we will break it down.

```
function stepToFive(number) {
  if (number > 5) {
                number -= 1;
  }
  if (number < 5) {
                number += 1;
  }
  return number;
}
```

This program takes a number as an argument. If the number is larger than 5, it subtracts one. If it's smaller than five it adds one. Let's get down and dirty and look at this piece by piece.

```
function stepToFive(number) {
```

This is similar to what we've seen before. We now have number following the function name. This is where we define our arguments for later use, which is similar to defining variables, except in this case the variables are only valid inside of the function.

```
  if (number > 5) {
```

If statements. If the condition is true, execute the code inside the curly brackets.

```
    number -= 1;
```

Assuming that JavaScript is your first language, you might not know what this means. This takes one off from the variable number. You can think of it as a useful shorthand for number = number - 1;.

```
    number += 1;
```

This is similar to the last one, except that it adds one instead of subtracting it.

```
  return number;
```

This returns the value of number from the function. This will be covered in more depth later on.

Here is an example of using this in a page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
 <head>
  <title>Some Page</title>
    <script type="text/javascript">

      function stepToFive(number) {
        if (number > 5) {
          number -= 1;
        }
        if (number < 5) {
          number += 1;
        }
        return number;
      }

    </script>
  </head>
  <body>
    <p>
      <script type="text/javascript">

        var num = stepToFive(6);
        alert(num);

      </script>
    </p>
  </body>
</html>
```

There are a few more things to cover here.

```
var num = stepToFive(6);
```

This is where the return statement in the function comes in handy. num here gets assigned the number 5, since that is what stepToFive will return when it's given an argument of 6.

Objects

An object is a structure of variables and some functions used to work with those variables. These include private variables, which should *only* be referenced or changed using the *methods* it provides. For example if a Date object's getFullYear() method returns an incorrect value, you can depend on the fact that somewhere in your script, the setFullYear() method has been used

to alter it.

The philosophy of object-oriented programming says that program code should be as modular as possible. Once you've written and tested a function, it should be possible to slot it into any program or script needing that kind of functionality and just expect it to work, because it's already been tried and tested on an earlier project.

The benefits of this approach are a shorter development time and easier debugging, because you're re-using program code that has already been proven. This 'black box' approach means that data goes into the Object and other data comes out of the Object, but what goes on inside it isn't something you need to concern yourself with.

An Object is a code structure that has its own private variables, and stores data that is isolated from outside interference. Only the Object itself can alter this data. The user works with the data through a set of public functions (called Methods) for accessing these private variables. It's probably most convenient to consider that an Object has its own set of functions that allows you to work with it in certain ways.

While JavaScript allows creating of objects, you will discover that access protection are not directly available in JavaScript; as such, it is possible to bypass the intent of the object-oriented programming by directly accessing fields or methods. To minimize the impact of these issues, you may want to ensure that methods are properly described and cover known situations of use - and to avoid directly accessing methods or fields that are designed to hold the internal state of the object.

Javascript provides a set of predefined Objects. For example: the document itself is an Object, with internal variables like 'title' and 'URL'.

The Date Object

Let's look at the Date Object. You can create a new object and assign it to a variable name using the *new* keyword:

```
var mydate = new Date();
```

The Date Object has a set of internal variables which hold the time, day, month, and year. It allows you to refer to it like a string, so you could for example pop-up the time that the variable was created. A pair of lines like this:

```
var myDate = new Date();
alert(myDate);
```

would display an alert box showing the current time and date, in universal time, like this:

```
Tue Jul 26 13:27:33 UTC+1200
2007
```

Even though it produced a string, the variable myDate is not actually one itself. An operator called *typeof* returns a string that indicates what type the variable is. These types are:

- boolean

- function
- number
- object
- string
- undefined

So the following code:

```
var myDate = new Date();
alert(typeof myDate);
```

would produce:

```
object
```

The Date Object stores a lot of information about the date, which are accessed using a certain predefined *method*. Some examples of these methods are:

- getFullYear()
- getMonth()
- getDate()
- getDay()
- getHours()
- getMinutes()
- getSeconds()

The following code shows the year and what type of object that information is.

```
var myDate = new Date();
var year = myDate.getFullYear();
alert(year + '\n' + typeof(year));
```

```
2014
number
```

Because information such as the year is private to the object, the only way we have to alter that information is to use a method provided by the Object for that purpose.

The above methods to get information from the Date object have matching methods that allow you to set them too.

- setFullYear()
- setMonth()
- setDate()
- setDay()
- setHours()
- setMinutes()
- setSeconds()

The following code will show one year, followed by a different year.

```
var myDate = new Date();
alert(myDate.getFullYear());
myDate.setFullYear(2008);
alert(myDate.getFullYear());
```

2014
2008

Defining New Objects

```
var site = {};
site.test = function (string) {
   alert("Hello World! " + string);
   site.string = string;
}
site.test("Boo!");
alert(site.string);
```

What this example does is:

- Define site as an empty object
- Add a method called test to the site object
- Call the test method with variable "Boo!"

The result is:

- An alert message with "Hello World! Boo!"
- site.string being defined as string
- An alert message with "Boo!".

this keyword

The this keyword allows a method to read and write the property variables of that instance of the object.

The following example uses an initial capital letter for PopMachine() to help indicate that the object needs to be created with the new keyword. You can use the new keyword with any function to create an object, but it will be much easier to keep track of what you're doing if you only use new with functions that are meant just for the purpose, and mark those functions by naming them with an initial capital letter.

```
function PopMachine() {
   this.quarters = 0;
   this.dollars = 0;
   this.totalValue = function () {
      var sum = this.quarters * 25 + this.dollars * 100;
      return sum;
   }
   this.addQuarters = function (increment) {
      this.quarters += increment;
   }
   this.addDollars = function (increment) {
```

```
        this.dollars += increment;
    }
}
function testPopMachine() {
    var popMachine = new PopMachine();
    popMachine.addQuarters(8);
    popMachine.addDollars(1);
    popMachine.addQuarters(-1);
    alert("Total in the cash register is: " + popMachine.totalValue());
}
testPopMachine();
```

Please notice that the above method is inefficient and will use way too much memory inside the browser, if you need to create a serious javascript class then you must first learn about prototypes.

Exceptions

In Javascript, errors are created by the Error object and its subclasses. To catch the error to prevent it from stopping your script, you need to enclose sections of your code with the try...catch block.

Errors have two important fields: Error.name - which is the name of the error, and Error.message - a human readable description of the error.

While you can throw any object you want as an exception, it's strongly recommended to throw an instance of the Error object.

Event Handling

Event Handlers

An event occurs when something happens in a browser window. The kinds of events that might occur are due to:

- A document loading
- The user clicking a mouse button
- The browser screen changing size

When a function is assigned to an event handler, that function is run when that event occurs.

A handler that is assigned from a script used the syntax '[element].[event] = [function];', where [element] is a page element, [event] is the name of the selected event and [function] is the name of the function that occurs when the event takes place.

For example:

```
document.onclick = clickHandler;
```

This handler will cause the function clickHandler() to be executed whenever the user clicks the mouse anywhere on the screen. Note that when an event handler is assigned, the function name does not end with parentheses. We are just pointing the event to the name of the function. The clickHandler function is defined like this:

```
function clickHandler(evt) {
  //some code here
}
```

By convention the event is represented by the variable 'evt'. In some browsers the event must be explicitly passed to the function, so as a precaution it's often best to include a conditional to test that the evt variable has been passed, and if it hasn't then to use an alternative method that works on those other browsers.:

```
function clickHandler(evt) {
    evt = evt || window.event;
    //some code here
}
```

Elements within a document can also be assigned event handlers. For example:

```
document.getElementsByTagName('a')[0].onclick = linkHandler;
```

This will cause the linkHandler() function to be executed when the user clicks the first link on the page.

Keep in mind that this style of handler assignment depends on the link's position inside the page. If another link tag is added before this one, it will take over the handler from the original

link. A best practice is to maintain the separation of code and page structure by assigning each link an identifier by using the id attribute.

```
<a id="faqLink" href="faq.html">Faq</a>
```

A handler assignment can then work regardless of where the element is positioned.

```
document.getElementById('faqLink').onclick = linkHandler;
```

Events are actions that can be detected by JavaScript, and the event object gives information about the event that has occurred. Sometimes we want to execute a JavaScript when an event occurs, such as when a user clicks a button. Events are normally used in combination with functions, and the function will not be executed before the event occurs! Javascript event handlers are divided into 2 types:

1. Interactive event handlers- depends on user interactin with the HTML page ex. Clicking a button
2. Non-Interactive event handlers-does not need user interaction. Ex. onload

Event Attributes

Below is the event attributes that can be inserted into different HTML elements to define event actions. IE: Internet Explorer, F: Firefox, O: Opera, W3C: W3C Standard.

Attribute	The event occurs when...	IE	F	O	W3C
onblur	An element loses focus	3	1	9	Yes
onchange	The content of a field changes	3	1	9	Yes
onclick	Mouse clicks an object	3	1	9	Yes
ondblclick	Mouse double-clicks an object	4	1	9	Yes
onerror	An error occurs when loading a document or an image	4	1	9	Yes
onfocus	An element gets focus	3	1	9	Yes
onkeydown	A keyboard key is pressed	3	1	No	Yes
onkeypress	A keyboard key is pressed or held down	3	1	9	Yes
onkeyup	A keyboard key is released	3	1	9	Yes
onload	A page or image is finished loading	3	1	9	Yes
onmousedown	A mouse button is pressed	4	1	9	Yes
onmousemove	The mouse is moved	3	1	9	Yes
onmouseout	The mouse is moved off an element	4	1	9	Yes
onmouseover	The mouse is moved	3	1	9	Yes

over an element
onmouseup
A mouse button is released 4 1 9 Yes
onresize
A window or frame is resized 4 1 9 Yes
onselect
Text is selected 3 1 9 Yes
onunload
The user exits the page 3 1 9 Yes

Mouse / Keyboard Attributes:

Property Description IE F O W3C
altKey
Returns whether or not the "ALT" 6 1 9 Yes
key was pressed when an event
was triggered
button
Returns which mouse button was 6 1 9 Yes
clicked when an event was triggered
clientX
Returns the horizontal coordinate of 6 1 9 Yes
the mouse pointer when an event was triggered
clientY
Returns the vertical coordinate of the 6 1 9 Yes
mouse pointer when an event was triggered
ctrlKey
Returns whether or not the "CTRL" key 6 1 9 Yes
was pressed when an event was triggered
metaKey
Returns whether or not the "meta" key 6 1 9 Yes
was pressed when an event was triggered
relatedTarget
Returns the element related to the No 1 9 Yes
element that triggered the event
screenX
Returns the horizontal coordinate of the 6 1 9 Yes
mouse pointer when an event was triggered
screenY
Returns the vertical coordinate of the mouse 6 1 9 Yes
pointer when an event was triggered
shiftKey
Returns whether or not the "SHIFT" key was 6 1 9 Yes
pressed when an event was triggered

Other Event Attributes:

Property Description IE F O W3C
bubbles
Returns a Boolean value that indicates No 1 9 Yes
whether or not an event is a bubbling event
cancelable
Returns a Boolean value that indicates No 1 9 Yes
whether or not an event can have
its default action prevented
currentTarget

Returns the element whose event listeners triggered the event	No	1	9	Yes	
Returns the element that triggered the event	No	1	9	Yes	
timeStamp					
Returns the time stamp, in milliseconds from the epoch (system start or event trigger)	No	1	9	Yes	

Standard event handlers

Attribute	Trigger
onabort	Loading of image was interrupted
onblur	Element loses focus
onchange	Element gets modified
onclick	Element gets clicked
ondblclick	Element gets double clicked
onerror	An error occurred loading an element
onfocus	An element received focus
onkeydown	A key was pressed when an element has focus
onkeypress	A keystroke was received by the element
onkeyup	A key was released when the element has focus
onload	An element was loaded
onmousedown	The mouse button was pressed on the element
onmousemove	The mouse pointer moves while inside the element
onmouseout	The mouse pointer was moved outside the element
onmouseover	The mouse pointer was moved onto the element
onmouseup	The mouse button was released on the element.
onreset	The form's reset button was clicked
onresize	The containing window or frame was resized
onselect	Text within the element was selected
onsubmit	A form is being submitted
onunload	The content is being unloaded (e.g. window being closed)
onscroll	The user scrolls (in any direction and with any means).

Event Handlers as HTML attributes

In HTML, JavaScript events can be included within any specified attribute - for example, a body tag can have an onload event:

```
<body onload="alert('Hello world!');">
```

The content of the HTML event attributes is JavaScript code that is interpreted when the event is triggered, and works very similarly to the blocks of JavaScript. This form of code is used when you want to have the JavaScript attached directly to the tag in question.

This type of technique is called inline JavaScript, and can be seen as being a less desirable technique than other unobtrusive JavaScript techniques that have previously been covered. The use of inline JavaScript can be considered to be similar in nature to that of using inline css, where HTML is styled by putting css in style attributes. This is a practice that is best avoided in favour of more versatile techniques.

Regular Expressions

JavaScript implements *regular expressions* (regex for short) when searching for matches within a string. In the following example, we are replacing the word *be* by the word *exist* in a text:

```
1. var shakespeareText = "To be or not to be? That is the question.";

2. var regularExpression = new RegExp("be", "g");

3. var spoiltShakespeareText = shakespeareText.replace(regularExpression, "exist");

4. alert(spoiltShakespeareText);
```

It will display:

To exist or not to exist? That is the question.

As with other scripting languages, this allows searching beyond a simple letter-by-letter match, and can even be used to parse strings in a certain format. Regular expressions most commonly appear in conjunction with the string.match() and string.replace() methods. A regular expression object can be created with the RegExp constructor. The first parameter is the pattern and the second the options:

```
1. var regularExpression = new RegExp("be", "g");
```

Alternatively, it can be delimited by the slash (*/*) character, and may have some options appended:

```
1. var regularExpression = /be/g;
```

Compatibility

JavaScript's set of regular expressions follows the extended set. While copying a Regex pattern from JavaScript to another location may work as expected, some older programs may not function as expected.

- In the search term, \1 is used to back reference a matched group, as in other implementations.
- In the replacement string, $1 is substituted with a matched group in the search, instead of \1.
    - Example: "abbc".replace(/(.)\1/g, "$1") => "abc"
- | is magic, \| is literal
- (is magic, \(is literal
- The syntaxes (?=...), (?!...), (?<=...), (?<!...) are not available.

JavaScript

Matching

1. string = "Hello world!".match(/world/);
2. stringArray = "Hello world!".match(/l/g); // Matched strings are returned in a string array
3. "abc".match(/a(b)c/)[1] => "b" // Matched subgroup is the second member (having the index "1") of the resulting array

Replacement

1. string = string.replace(/expression without quotation marks/g, "replacement");
2. string = string.replace(/escape the slash in this\/way/g, "replacement");
3. string = string.replace(...).replace (...). replace(...);

Test

1. if (string.match(/regexp without quotation marks/)) {

Modifiers

Modifier	Note
g	Global. The list of matches is returned in an array.
i	Case-insensitive search
m	Multiline. If the operand string has multiple lines, ^ and $ match the beginning and end of each line within the string, instead of matching the beginning and end of the whole string only. • "a\nb\nc".replace(/^b$/g,"d") => "a\nb\nc" • "a\nb\nc".replace(/^b$/gm,"d") => "a\nd\nc"

Operators

Operator	Effect
\b	Matches boundary of a word.
\w	Matches an alphanumeric character, including "_".
\W	Negation of \w.
\s	Matches a whitespace character (space, tab, newline, formfeed)
\S	Negation of \s.

\d	Matches a digit.
\D	Negation of \d.

Function call

For complex operations, a function can process the matched substrings. In the following code, we are capitalizing all the words. It can't be done by a simple replacement as each letter to capitalize is a different character:

```
1. var capitalize = function(matchobj) {
2.  var group1 = matchobj.replace(/^(\W)[a-zA-Z]+$/g, "$1");
3.  var group2 = matchobj.replace(/^\W([a-zA-Z])[a-zA-Z]+$/g, "$1");
4.  var group3 = matchobj.replace(/^\W[a-zA-Z]([a-zA-Z]+)$/g, "$1");
5.  return group1 + group2.toUpperCase() + group3;
6. };
7. var shakespeareText = "To be or not to be? That is the question.";
8. var spoiltShakespeareText = shakespeareText.replace(/\W[a-zA-Z]+/g, capitalize);
9. alert(spoiltShakespeareText);
```

It will display:

To Be Or Not To Be? That Is The
Question.

The function is called for each substring. Here is the signature of the function:

function (*<matchedSubstring>[, <capture1>, ...<captureN>, <indexInText>, <entireText>]*) {

    ...
    return *<stringThatWillReplaceInText>*;

}

- The first parameter is the substring that matches the pattern,
- The next parameters are the captures in the substrings. There are as many parameters as there are captures,
- The next parameter is the index of the beginning of the substring starting from the beginning of the text,
- The last parameter is a remainder of the entire text,

- The return value will be put in the text instead of the matching substring.

Optimization

JavaScript Optimization

Optimization Techniques

- High Level Optimization
  - Algorithmic Optimization (Mathematical Analysis)
  - Simplification
- Low Level Optimization
  - Loop Unrolling
  - Strength Reduction
  - Duff's Device
  - Clean Loops
- External Tools & Libraries for speeding/optimizing/compressing JavaScript code

Common Mistakes and Misconceptions

String concatenation

Strings in JavaScript are immutable objects. This means that once you create a string object, to modify it, another string object must theoretically be created.

Now, suppose you want to perform a ROT-13 on all the characters in a long string. Supposing you have a rot13() function, the obvious way to do this might be:

```
var s1 = "the original string";
var s2 = "";

for(i=0; i < s1.length; i++) {
  s2 += rot13(s1.charAt(i));
}
```

Especially in older browsers like Internet Explorer 6, this will be very slow. This is because, at each iteration, the entire string must be copied before the new letter is appended.

One way to make this script faster might be to create an array of characters, then join it:

```
var s1 = "the original string";
var a2 = new Array(s1.length);
var s2 = "";

for(i=0; i < s1.length; i++) {
  a2[i] = rot13(s1.charAt(i));
}
s2 = a2.join('');
```

Internet Explorer 6 will run this code faster. However, since the original code is so obvious and easy to write, most modern browsers have improved the handling of such concatenations. On some browsers the original code may be faster than this code.

A second way to improve the speed of this code is to break up the string being written to. For instance, if this is normal text, a space might make a good separator:

```
var s1 = "the original string";
var c;
var st = "";
var s2 = "";

for(i=0; i < s1.length; i++) {
  c = rot13(s1.charAt(i));
  st += c;
  if(c == " ") {
    s2 += st;
    st = "";
  }
}
s2 += st;
```

This way the bulk of the new string is copied much less often, because individual characters are added to a smaller temporary string.

A third way to really improve the speed in a for loop, is to move the [array].length statement outside the condition statement. In face, every occurrence, the [array].length will be re-calculate For a two occurences loop, the result will not be visible, but (for example) in a five thousand occurence loop, you'll see the difference. It can be explained with a simple calculation :

```
// we assume that myArray.length is 5000
for(x=0;x<myArray.lengh;x++){
// doing some stuff
}
```

"x=0" is evaluated only one time, so it's only one operation.

"x<myArray.length" is evaluted 5000 times, so it is 10 000 operations (myArray.length is an operation and compare myArray.length with x, is another operation).

"x++" is evaluated 5000 times, so it's 5000 operations.

There is a total of 15 001 operation.

```
// we assume that myArray.length is 5000
for(x=0, l=myArray.lengh; x<l; x++){
// doing some stuff
}
```

"x=0" is evaluated only one time, so it's only one operation.

"l=myArray.length" is evaluated only one time, so it's only one operation.

"x<l" is evaluted 5000 times, so it is 5000 operations (l with x, is one operation).

"x++" is evaluated 5000 times, so it's 5000 operations.

There is a total of 10002 operation.

So, in order to optimize your for loop, you need to make code like this :

```
var s1 = "the original string";
var c;
var st = "";
var s2 = "";

for(i=0, l = s1.length; i < l; i++) {
  c = rot13(s1.charAt(i));
  st += c;
  if(c == " ") {
    s2 += st;
    st = "";
  }
}
s2 += st;
```

Debugging

JavaScript Debuggers

Firebug

- Firebug is a powerful extension for Firefox that has many development and debugging tools including JavaScript debugger and profiler.

Venkman JavaScript Debugger

- Venkman JavaScript Debugger (for Mozilla based browsers such as Netscape 7.x, Firefox/Phoenix/Firebird and Mozilla Suite 1.x)
- Introduction to Venkman
- Using Breakpoints in Venkman

Internet Explorer debugging

- Microsoft Script Debugger (for Internet Explorer) The script debugger is from the Windows 98 and NT era. It has been succeeded by the Developer Toolbar
- Internet Explorer Developer Toolbar
- Microsofts Visual Web Developer Express is Microsofts free version of the Visual Studio IDE. It comes with a JS debugger. For a quick summary of its capabilities see [3]
- Internet Explorer 8 has a firebug-like web development tool by default (no add-on) which can be accessed by pressing F12. The web development tool also provides the ability to switch between the IE8 and IE7 rendering engines.

Safari debugging

Safari includes a powerful set of tools that make it easy to debug, tweak, and optimize a website for peak performance and compatibility. To access them, turn on the Develop menu in Safari preferences. These include Web Inspector, Error Console, disabling functions, and other developer features. The Web Inspector gives you quick and easy access to the richest set of development tools ever included in a browser. From viewing the structure of a page to debugging JavaScript to optimizing performance, the Web Inspector presents its tools in a clean window designed to make developing web applications more efficient. To activate it, choose Show Web Inspector from the Develop menu. The Scripts pane features the powerful JavaScript Debugger in Safari. To use it, choose the Scripts pane in the Web Inspector and click Enable Debugging. The debugger cycles through your page’s JavaScript, stopping when it encounters exceptions or erroneous syntax. The Scripts pane also lets you pause the JavaScript, set breakpoints, and evaluate local variables.[1]

JTF: Javascript Unit Testing Farm

- JTF is a collaborative website that enables you to create test cases that will be tested by all browsers. It's the best way to do TDD and to be sure that your code will work well on all browsers.

jsUnit

- jsUnit

built-in debugging tools

Some people prefer to send debugging messages to a "debugging console" rather than use the alert() function[4][5][6]. Following is a brief list of popular browsers and how to access their respective consoles/debugging tools.

- Firefox: Ctrl+Shift+K opens an error console.
- Opera (9.5+): Tools >> Advanced >> Developer Tools opens Dragonfly.
- Chrome: Ctrl+Shift+J opens chrome's "Developer Tools" window, focused on the "console" tab.
- Internet Explorer: F12 opens a firebug-like web development tool that has various features including the ability to switch between the IE8 and IE7 rendering engines.
- Safari: Cmd+Alt+C opens the WebKit inspector for Safari.

Common Mistakes

- Carefully read your code for typos.
- Be sure that every "(" is closed by a ")" and every "{" is closed by a "}".
- Trailing commas in Array and Object declarations will throw an error in Microsoft Internet Explorer but not in Gecko-based browsers such as Firefox.

```
// Object
var obj = {
  'foo'   : 'bar',
  'color' : 'red', //trailing comma
};

// Array
var arr = [
  'foo',
  'bar', //trailing comma
];
```

- Remember that JavaScript is case sensitive. Look for case related errors.
- Don't use Reserved Words as variable names, function names or loop labels.
- Escape quotes in strings with a "\" or the JavaScript interpreter will think a new string is being started, i.e:

  ~~alert('He's eating food');~~ should be

alert('He\'s eating food'); or
alert("He's eating food");

- When converting strings to numbers using the parseInt function, remember that "08" and "09" (e.g. in datetimes) indicate an octal number, because of the prefix zero. Using parseInt using a radix of 10 prevents wrong conversion. var n = parseInt('09',**10**);
- Remember that JavaScript is platform independent, but is not browser independent. Because there are no properly enforced standards, there are functions, properties and even objects that may be available in one browser, but not available in another, e.g. Mozilla / Gecko Arrays have an indexOf() function; Microsoft Internet Explorer does not.

Debugging Methods

Debugging in Javascript doesn't differ very much from debugging in most other programming languages. See the article at Computer programming/debugging.

Following Variables as a Script is Running

The most basic way to inspect variables while running is a simple alert() call. However some development environments allow you to step through your code, inspecting variables as you go. These kind of environments may allow you to change variables while the program is paused.

Browser Bugs

Sometimes the browser is buggy, not your script. This means you must find a workaround.

Browser bug reports

browser-dependent code

Some advanced features of Javascript don't work in some browsers.

Too often our first reaction is: Detect which browser the user is using, then do something the cool way if the user's browser is one of the ones that support it. Otherwise skip it.

Instead of using a "browser detect", a much better approach is to write "object detection" Javascript to detect if the user's browser supports the particular object (method, array or property) we want to use.[7] [8]

To find out if a method, property, or other object exists, and run code if it does, we write code like this:

```
var el = null;
if (document.getElementById) {
   // modern technique
   el = document.getElementById(id);
} else if (document.all) {
   // older Internet Explorer technique
   el = document.all[id];
} else if (document.layers) {
```

```
    // older Netscape web browser technique
    el = document.layers[id];
}
```

For further reading

- "Javascript Debugging" by Ben Bucksch

1. http://www.apple.com/safari/features.html#developer

DHTML

DHTML (**D**ynamic **HTML**) is a combination of JavaScript, CSS and HTML.

alert messages

```
<script type="text/javascript">
    alert('Hello World!');
</script>
```

This will give a simple alert message.

```
<script type="text/javascript">
    prompt('What is your name?');
</script>
```

This will give a simple prompt message.

```
<script type="text/javascript">
confirm('Are you sure?');
</script>
```

This will give a simple confirmation message.

Javascript Button and Alert Message Example:

Sometimes it is best to dig straight in with the coding. Here is an example of a small piece of code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
 <head>
  <title>"THE BUTTON" - Javascript</title>
    <script type="text/javascript">
      x = 'You have not pressed "THE BUTTON"'
      function bomb() {
        alert('O-GOD NOOOOO, WE ARE ALL DOOMED!!');
        alert('10');
        alert('9');
```

```
      alert('8');
      alert('7');
      alert('6');
      alert('5');
      alert('4');
      alert('3');
      alert('2');
      alert('1');
      alert('!BOOM!');
      alert('Have a nice day. :-)');
      x = 'You pressed "THE BUTTON" and I told you not to!';
    }
  </script>
  <style type="text/css">
    body {
      background-color:#00aac5;
      color:#000
    }
  </style>
 </head>
 <body>
  <div>
    <input type="button" value="THE BUTTON - Don't Click It" onclick="bomb()"><br />
    <input type="button" value="Click Here And See If You Have Clicked "THE BUTTON"" onclick="alert(x)">
  </div>
  <p>
    This script is dual-licensed under both, <a
href="http://www.wikipedia.org/wiki/GNU_Free_Documentation_License">GFDL</a> and <a href="GNU General
Public License">GPL</a>. See <a href="http://en.wikibooks.org/wiki/JavaScript">Wikibooks</a>
  </p>
 </body>
</html>
```

What has this code done? Well when it loads it tells what value the variable 'x' should have. The next code snippet is a function that has been named "bomb". The body of this function fires some alert messages and changes the value of 'x'.

The next part is mainly HTML with a little javascript attached to the INPUT tags. "onclick" property tells its parent what has to be done when clicked. The bomb function is assigned to the first button, the second button just shows an alert message with the value of x.

Javascript if() - else Example

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
 <head>
  <title>The Welcome Message - Javascript</title>
  <script type="text/javascript">
    function wlcmmsg() {
      name = prompt('What is your name?', '');
      correct = confirm('Are you sure your name is ' + name + ' ?');
      if (correct == true) {
        alert('Welcome ' + name);
      } else {
```

```
        wlcmmsg();
      }
    }
  </script>
  <style type="text/css">
    body {
      background-color:#00aac5;
      color:#000
    }
  </style>
 </head>
 <body onload="wlcmmsg()" onunload="alert('Goodbye ' + name)">
  <p>
   This script is dual-licensed under both, <a
href="http://www.wikipedia.org/wiki/GNU_Free_Documentation_License">GFDL</a> and <a href="GNU General
Public License">GPL</a>. See <a href="http://textbook.wikipedia.org/wiki/Programming:Javascript">Wikibooks</a>
  </p>
 </body>
</html>
```

Two Scripts

We are going back to the first example. But adding more to the script by also adding a different welcome message. This time a person is made to enter a name. They are also asked if they want to visit the site. Some CSS has also been added to the button.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
 <head>
  <title>"THE BUTTON" - Javascript</title>
  <script type="text/javascript">
    // global variable x
    x = 'You have not pressed "THE BUTTON"';

    function bomb() {
      alert('O-GOD NOOOOO, WE ARE ALL DOOMED!!');
      alert('10');
      alert('9');
      alert('8');
      alert('7');
      alert('6');
      alert('5');
      alert('4');
      alert('3');
      alert('2');
      alert('1');
      alert('!BOOM!');
      alert('Have a nice day. :-)');
      x = 'You pressed "THE BUTTON" and I told you not too!';
    }
  </script>
  <style type="text/css">
    body {
      background-color:#00aac5;
```

```
      color:#000
    }
  </style>
 </head>
 <body onload="welcome()">
  <script type="text/javascript">
   function welcome() {
    var name = prompt('What is your name?', '');
    if (name == "" || name == "null") {
     alert('You have not entered a name');
     welcome();
     return false;
    }
    var visit = confirm('Do you want to visit this website?')
    if (visit == true) {
     alert('Welcome ' + name);
    } else {
     window.location=history.go(-1);
    }
   }
  </script>
  <div>
   <input type="button" value="THE BUTTON - Don't Click It" onclick="bomb()" STYLE="color: #ffdd00;
background-color: #ff0000"><br>
   <input type="button" value="Click Here And See If You Have Clicked "THE BUTTON"" onclick="alert(x)">
  </div>
  <p>
   This script is dual-licensed under both, <a
href="http://www.wikipedia.org/wiki/GNU_Free_Documentation_License">GFDL</a> and <a href="GNU General
Public License">GPL</a>. See <a
href="http://textbook.wikipedia.org/wiki/Programming:Javascript">Wikibooks</a>,
  </p>
 </body>
</html>
```

Simple Calculator

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
 <head>
  <title>Calculator</title>
  <script type="text/javascript">
  function multi() {
   var a = document.Calculator.no1.value;
   var b = document.Calculator.no2.value;
   var p = (a*b);
   document.Calculator.product.value = p;
  }

  function divi() {
   var d = document.Calculator.dividend.value;
   var e = document.Calculator.divisor.value;
   var q = (d/e);
   document.Calculator.quotient.value = q;
```

```
    }

    function circarea() {
      var r = document.Calculator.radius.value;
      pi =
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316 5;
      var a = pi*(r*r);
      document.Calculator.area.value = a;
      var c = 2*pi*r;
      document.Calculator.circumference.value = c;
    }
    </script>
    <style type="text/css">
      body {
        background-color:#00aac5;
        color:#000
      }

      label {
        float:left;
        width:7em
      }
    </style>
  </head>
  <body>
    <h1>Calculator</h1>
    <form name="Calculator" action="">
      <fieldset>
        <legend>Multiply</legend>
        <input type="text" name="no1"> × <input type="text" name="no2">
        <input type="button" value="=" onclick="multi()">
        <input type="text" name="product">
      </fieldset>
      <fieldset>
        <legend>Divide</legend>
        <input type="text" name="dividend"> ÷ <input type="text" name="divisor">
        <input type="button" value="=" onclick="divi()">
        <input type="text" name="quotient">
      </fieldset>
      <fieldset>
        <legend>Area and Circumfrence of Circle</legend>
        <p>(Uses pi to 240 d.p)</p>
        <div>
          <label for="radius">Type in radius</label> <input type="text" name="radius" id="radius" value="">
        </div>
        <div>
          <input type="button" value="=" onclick="circarea()">
        </div>
        <div>
          <label for="area">Area</label> <input type="text" name="area" id="area" value="">
        </div>
        <div>
          <label for="circumference">Circumference</label> <input type="text" name="circumference"
```

```
id="circumference" value="">
        </div>
      </fieldset>
    </form>
    <p>Licensed under the <a href="http://www.gnu.org/licenses/gpl.html">GNU GPL</a>.</p>
  </body>
</html>
```

Finding Elements

The most common method of detecting page elements in the DOM is by the **document.getElementById(id)** method.

Simple Use

Let's say, on a page, we have:

```
<div id="myDiv">content</div>
```

A simple way of finding this element in Javascript would be:

```
var myDiv = document.getElementById("myDiv"); // Would find the DIV element by its ID, which in this case is 'myDiv'.
```

Use of getElementsByTagName

Another way to find elements on a web page is by the **getElementsByTagName(name)** method. It returns an array of all *name* elements in the node.

Let's say, on a page, we have:

```
<div id="myDiv">
   <p>Paragraph 1</p>
   <p>Paragraph 2</p>
   <h1>An HTML header</h1>
   <p>Paragraph 3</p>
</div>
```

Using the getElementsByTagName method we can get an array of all <p> elements inside the div:

```
var myDiv = document.getElementById("myDiv"); // get the div
var myParagraphs = myDiv.getElementsByTagName('P'); //get all paragraphs inside the div

// for example you can get the second paragraph (array indexing starts from 0)
var mySecondPar = myParagraphs[1]
```

Adding Elements

Basic Usage

Using the **Document Object Module** we can create basic HTML elements. Let's create a div.

```
var myDiv = document.createElement("div");
```

What if we want the div to have an ID, or a class?

```
var myDiv = document.createElement("div");
myDiv.id = "myDiv";
myDiv.class = "main";
```

And we want it added into the page? Let's use the DOM again...

```
var myDiv = document.createElement("div");
myDiv.id = "myDiv";
myDiv.class = "main";
document.documentElement.appendChild(myDiv);
```

Further Use

So let's have a simple HTML page...

```
<html>
<head>
</head>
<body bgcolor="white" text="blue">
<h1> A simple Javascript created button </h1>
<div id="button"></div>
</body>
</html>
```

Where the div which has the id of button, let's add a button.

```
myButton = document.createElement("input");
myButton.type = "button";
myButton.value = "my button";
placeHolder = document.getElementById("button");
placeHolder.appendChild(myButton);
```

All together the HTML code looks like:

```
<html>
<head>
</head>
<body bgcolor="white" text="blue">
<h1> A simple Javascript created button </h1>
<div id="button"></div>
</body>
```

```
<script>
myButton = document.createElement("input");
myButton.type = "button";
myButton.value = "my button";
placeHolder = document.getElementById("button");
placeHolder.appendChild(myButton);
</script>
</html>
```

The page will now have a button on it which has been created via Javascript.

Changing Elements

In JavaScript you can change elements by using the following syntax:

element.attribute="new value"

Here, the "src" attribute of an image is changed so when the script is called, it changes the picture from "myPicture.jpg" to "otherPicture.jpg".

```
//The Html
<img id="example" src="myPicture.jpg">
//The JavaScript
document.getElementById("example").src="otherPicture.jpg";
```

In order to change an element, you use its argument name for the value you wish to change. For example, let's say we have a button, and we wish to change its value.

```
<input type="button" id="myButton" value="I'm a button!">
```

Later on in the page, with JavaScript, we could do the following to change that button's value:

```
myButton = document.getElementById("myButton"); //searches for and detects the input element from the 'myButton' id
myButton.value = "I'm a changed button"; //changes the value
```

To change the type of input it is (button to text, for example) then use:

```
myButton.type = "text"; //changes the input type from 'button' to 'text'.
```

Another way to change or create an attribute is to use a method like element.setAttribute("attribute","value") or element.createAttribute("attribute","value"). Use "setAttribute()" to change a attribute that has been defined before.

```
//The Html
<div id="div2"></div> //Make a div with an id of div2 (we also could have made it with JavaScript)
//The Javascript
var e = document.getElementById("div2"); //Get the element
e.setAttribute("id","div3"); //Change id to div3
```

But use "createAttribute()" if you want to set a value that hasn't been defined before.

```
var e = document.createElement('div'); //Make a div element (we also could have made it with Html)
e.createAttribute("id","myDiv"); //Set the id to "myDiv"
```

Removing Elements

In Javascript, an element can only be deleted from its parent. To delete one, you have to get the element, then find its parent, and delete it using the deleteChild method.[1]

For example, in a HTML document that looks like

```
<div id="parent">
   <p id="child">I'm a child!</p>
</div>
```

The code you would need to delete the element with the ID "child" would be

```
// get elements
var child = document.getElementById("child");
var parent = document.getElementById("parent");

// delete child
parent.removeChild(child);
```

Instead of getting the parent element manually, you can use the parentNode property of the child to find its parent automatically. The code for this on the above HTML document would look like

```
// get the child element node
var child = document.getElementById("child");

// remove the child element from the document
child.parentNode.removeChild(child);
```

1. StackOverflow: Javascript: remove element by id

Code Structuring

Links

Links

Featured weblinks:

- JavaScript portal at developer.mozilla.org
- JavaScript Reference at developer.mozilla.org
- JavaScript Guide at developer.mozilla.org
- Gecko DOM Reference at developer.mozilla.org

- JavaScript Reference at msdn.microsoft.com

- Wikipedia:JavaScript
- Wikipedia:ECMAScript
- Wikipedia:JavaScript engine

- JavaScript Tutorial at w3schools.com
- JavaScript Reference at w3schools.com
- About: Focus on JavaScript from Stephen Chapman at javascript.about.com

- ecmascript.org
- ecma-international.org
- http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Discussion forums, bulletin boards:

- HTML, CSS and JavaScript at coderanch.com
- JavaScript Workshop forums at jsworkshop.com
- Forum: Client-side technologies at webxpertz.net

More web sites:

- JavaScript tutorials at webreference.com
- Videos from w:Douglas Crockford on JavaScript
- JavaScript at epanorama.net
- JavaScript Tutorials at pickatutorial.com
- JavaScript Essentials at techotopia.com - An online JavaScript book designed to provide web developers with everything they need to know to create rich, interactive and dynamic

web pages using JavaScript.

- JavaScript Tutorials at yourhtmlsource.com
- www.quirksmode.org - over 150 useful pages for CSS and Javascript tips & cross-browser compatibility matrices.
- Wiki: I wanna Learn JavaScript at c2.com - A list of links to web resources on JavaScript
- Unobtrusive JavaScript at onlinetools.org - a guide on how to write JavaScript so that your site degrades gracefully (i.e., if the browser does not support or has turned off JavaScript, your site is still usable).

Useful Software Tools

A list of useful tools for JavaScript Programmers.

Editors / IDEs

- Sublime Text: One of the most used editors for HTML/CSS/JavaScript editing
- Notepad++: A Great tool for editing any kind of code, includes syntax highlighting for many programming languages.
- Programmers' Notepad: A general tool for programming many languages.
- Scripted: An open source browser-based editor by Spring Source
- Adobe_Brackets: Another browser-based editor by Adobe
- Eclipse: the Eclipse IDE includes an editor and debugger for JavaScript
- Web Storm or IntelliJ IDEA: both IDEs include and editor and debugger for JavaScript, IDEA also includes a Java development platform

Engines and other tools

- List of ECMAScript engines
- JSLint: static code analysis for JavaScript
- jq - " 'jq' is like sed for JSON data "
- List of Really Useful Free Tools For JavaScript Developers