JavaScript/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) that remains the same during the lifespan of a running program in Java, and have a dynamic type (Number or String for example) that 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 that 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.

JavaScript/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?

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?

JavaScript/Placing the code

Contents

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

while a `script` element that contains inline JavaScript looks like:

```
<script>
  // 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 that can potentially be used in more than one page, and also to separate content from behaviour.

Inline JavaScript[edit]

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[edit]

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 that 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 onward, those browsers were able to interpret JavaScript.[1] Any modern browser that doesn't support JavaScript will still recognize the <script> tag and not display it to the user.

Inline XHTML JavaScript

In XHTML, the method is somewhat different:

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

Note that the `<![CDATA[` tag is commented out. The `//` prevents the browser from mistakenly interpreting the `<![CDATA[` as a JavaScript statement (that 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 method is considered a best practice. This is because it separates a page's behavior (JavaScript) from its content (HTML), and it makes it easier to update code. If several pages all link to the same JavaScript file, you only have to change the code in one place.

Add `src="script.js"` to the opening `script` tag. This means that the JavaScript code for this page will be located in a file called "script.js" that is in the same directory as the web page. If the JavaScript file is located somewhere else, you must change the `src` attribute to that path. For example, if your JavaScript file is located in a directory called "js", your `src` would be "js/script.js".

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 src="script.js"></script>
</head>
<body>
<!-- HTML code here -->
</body>
</html>
```

There are however some best practices for speeding up your web site [2] 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 src="script.js"></script>
</body>
</html>
```

JavaScript/Lexical structure

Contents

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(event){
event=event||window.event;
var charCode=event.charCode||event.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(event) {
```

```
  event = event || window.event;
  var charCode = event.charCode || event.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 called *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`

`1A2B3C` is an invalid identifier, as it starts with a number.

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

JavaScript/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.

Contents

Variable declaration

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

```
var c;
```

Doing so is obligatory, if the code includes the string comment `"use strict;"`. 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 that 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."

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 two possible values, true or false.

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

Numeric types

You can use an 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[edit]

Main page: JavaScript/Arrays

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.

Scope

In JavaScript, the scope is the current context of the code. It refers to the accessability of functions and variables, and their context. There exists a *global* and a *local* scope. The understanding of scope is crucial to writing good code. Whenever the code accesses `this`, it accesses the object that "owns" the current scope.

Global scope

An entity like a function or a variable has **global scope**, if it is accessible from everywhere in the code.

```
var a = 99;

function hello() {
  alert("Hello, " + a + "!");
}
hello();    // prints the string "Hello, 99!"
alert(a);   // prints the number 99
console.log("a = " + a); // prints "a = 99" to the console of the browser
```

Here, the variable `a` is in global scope and accessible both in the main code part and the function `hello()` itself. If you want to debug your code, you may use the `console.log(...)`command that outputs to the console window in your browser. This can be opened under the Windows OS with the F12 key.

Local scope

A **local scope** exists when an entity is defined in a certain code part, like a function.

```
var a = 99;

function hello() {
  var x = 5;
  alert("Hello, " + (a + x) + "!");
}
hello();    // prints the string "Hello, 104!"
alert(a);   // prints the number 99
alert(x);   // throws an exception
```

If you watch the code on a browser (on Google Chrome, this is achieved by pressing F12), you will see an **`Uncaught ReferenceError: x is not defined`** for the last line above. This is because `x` is defined in the local scope of the function hello and is not accessible from the outer part of the code.

JavaScript/Numbers

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

Contents

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

`ceil(float)`

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

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

`floor(float)`

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

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

`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
```

random()

Generates a pseudo-random number.

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

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 or functions:

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

JavaScript/Strings

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

Contents

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.

`concat(text)`

The `concat()` function joins two strings.

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

`length`

Returns the length as an integer.

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

`indexOf`

Returns the first occurrence of a string inside of itself, starting with 0. If the search string cannot be found, -1 is returned. The indexOf() method is case sensitive.

```
var foo = "Hello, World! How do you do?";
alert(foo.indexOf(' '));      // 6
```

```
var hello = "Hello world, welcome to the universe.";
alert(hello.indexOf("welcome"));      // 13
```

lastIndexOf

Returns the last occurrence of a string inside of itself, starting with index 0.. If the search string cannot be found, -1 is returned.

```
var foo = "Hello, World! How do you do?";
alert(foo.lastIndexOf(' '));   // 24
```

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.

slice(start[, end])

Slice extracts characters from the *start* position.

```
"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 `slice`.

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

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

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

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"
```

toLowerCase()

This function returns the current string in lower case.

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

toUpperCase()

This function returns the current string in upper case.

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

JavaScript/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, 12, 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:

- 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.
- getFullYear(): Retrieves the full 4-digit year within the Date object.
- getMonth(): Returns the current month. [0 - 11]
- parse(text): Reads the string *text*, and returns the number of milliseconds since January 1, 1970.
- setFullYear(year): Stores 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.

JavaScript/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]`.)

Contents

Overview

JavaScript arrays at a glance:

```
animals = ["cat", "dog", "tiger"]          // Initialization
fruits = [["apple", "orange"], ["pear"]]   // Initialization of a nested
array
cat = animals[0]                           // Indexed access
apple = fruits[0][0]                       // Indexed access, nested
animals[0] = "fox"                         // Write indexed access

for (var i = 0; i < animals.length; ++i)
  item = animals[i]                        // For each loop

animals = animals.concat("mouse")          // Append
animals = animals.concat(["horse", "ox"])  // Expand
animals.pop()                              // Yields "ox", removing it from
animals
animals.push("ox")                         // Push ox back to the end
```

```
animals.shift()                          // Yields "fox", removing it from
animals
animals.unshift("fox")                   // Place fox back to the beginning
mytext = [1, 2, 3].join("-")             // Yields "1-2-3" via join
items = mytext.split("-")                // Splits
```

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"
```

JavaScript/Regular expressions

Contents

Overview

JavaScript implements *regular expressions* (regex for short) when searching for matches within a string. 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.

Unlike strings, regular expressions are delimited by the slash (*/*) character, and may have some options appended.

Regular expressions most commonly appear in conjunction with the string.match() and string.replace() methods.

At a glance, by example:

```
strArray = "Hello world!".match(/world/); // Singleton array; note the slashes
strArray = "Hello!".match(/l/g); // Matched strings are returned in a string array
"abc".match(/a(b)c/)[1] === "b" // Matched subgroup is the 2nd item (index 1)
str1 = "Hey there".replace(/Hey/g, "Hello");
str2 = "N/A".replace(/\//g, ","); // Slash is escaped with \
str3 = "Hello".replace(/l/g, "m").replace(/H/g, "L").replace(/o/g, "a"); // Pile
if (str3.match(/emma/)) { console.log("Yes"); }
if (str3.match("emma")) { console.log("Yes"); } // Quotes work as well
"abbc".replace(/(.)\1/g, "$1") === "abc" // Backreference
```

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 (?=...), (?!...), (?<=...), and (?<!...) are not available.

Examples

- Matching
    - string = "Hello world!".match(/world/);
    - stringArray = "Hello world!".match(/l/g); // Matched strings are returned in a string array
    - "abc".match(/a(b)c/)[1] => "b" // Matched subgroup is the second member (having the index "1") of the resulting array
- Replacement
    - string = string.replace(/expression without quotation marks/g, "replacement");
    - string = string.replace(/escape the slash in this\/way/g, "replacement");
    - string = string.replace(...).replace (...). replace(...);
- Test
    - if (string.match(/regexp without quotation marks/)) {

Modifiers

Single-letter modifiers:

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:

```
var capitalize = function(matchobj) {
  var group1 = matchobj.replace(/^(\W)[a-zA-Z]+$/g, "$1");
  var group2 = matchobj.replace(/^\W([a-zA-Z])[a-zA-Z]+$/g, "$1");
  var group3 = matchobj.replace(/^\W[a-zA-Z]([a-zA-Z]+)$/g, "$1");
  return group1 + group2.toUpperCase() + group3;
};

var classicText = "To be or not to be?";

var changedClassicText = classicText.replace(/\W[a-zA-Z]+/g, capitalize);

console.log(changedClassicText==="To Be Or Not To Be?");
```

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.

JavaScript/Operators

Contents

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 seven 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 that 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.

Arithmetic	Logical	Shift
+=	&=	>>=
-=	|=	<<=
*=	^=	>>>=
/=		
%=		

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

```
var element = document.getElementsByTagName('h2');
var i;
for (i = 0; i < element .length; i += 1) {
  // do something with element [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 four lines all perform the same task:

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

Pre and post-increment operators

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

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

Be careful when using == and !=, as they may ignore the type of one of the terms 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 its 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(event) {
    event = event || window.event;
    var target = event.target || event.srcElement;
    if (target && target.nodeType === 1 && target.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, NaN, 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 target = (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.

Instead, it is better to 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;
}
```

delete

`delete x` unbinds `x`.

new

`new cl` creates a new object of type *cl*. The *cl* operand must be a constructor function.

instanceof

`o instanceof c` tests whether `o` is an object created by the constructor `c`.

typeof

`typeof x` returns a string describing the type of `x`. Following values may be returned:[1]

Type	returns
boolean	"boolean"
number	"number"
string	"string"
function	"function"
undefined	"undefined"
null	"object"
others	"object"

JavaScript/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.

Most of the statements enlisted below are so-called conditional statements that 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 nor an empty string is treated as true.

Contents

Conditional statements

if

The `if` statement is straightforward — 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.

A second version of this loop is the `for .. in` statement that has following form:

```
for (element in object) {
  // …
}
```

The order of the got elements is arbitrary. It should not be used when the object is of Array type

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, its 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.

Omitting the `break` can be used to test for more than one value at a time:

```
switch(i) {
case 1:
case 2:
case 3:
  // …
  break;
case 4:
  // …
  break;
default:
  // …
  break;
}
```

In this case the program will run the same code in case `i` equals 1, 2 or 3.

with

The `with` statement is used to extend the scope chain for a block[1] and has the following syntax:

```
with (expression) {
  // statement
}
```

Pros

The with statement can help to

- reduce file size by reducing the need to repeat a lengthy object reference, and
- relieve the interpreter of parsing repeated object references.

However, in many cases, this can be achieved by using a temporary variable to store a reference to the desired object.

Cons

The with statement forces the specified object to be searched first for all name lookups. Therefore

- all identifiers that aren't members of the specified object will be found more slowly in a 'with' block and should only be used to encompass code blocks that access members of the object.
- `with` makes it difficult for a human or a machine to find out which object was meant by searching the *scope chain*.
- Used with something else than a plain object, `with` may not be forward-compatible.

Therefore, the use of the with statement is not recommended, as it may be the source of confusing bugs and compatibility issues.

Example

```
var area;
var r = 10;

with (Math) {
  a = PI*r*r;       // == a = Math.PI*r*r
  x = r*cos(PI);    // == a = r*Math.cos(Math.PI);
  y = r*sin(PI/2);  // == a = r*Math.sin(Math.PI/2);
}
```

JavaScript/Functions

Contents

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.

A JavaScript function is a code-block that can be reused. The function can be called via an event, or by manual calling.

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:

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

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

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

Examples

Basic example

```
function myFunction(string) {
  alert(string);
  document.innerHTML += string;
}
myFunction("hello");
```

The example would first:

- Define the myFunction function
- Call the myFunction function with argument "hello"

The result:

- An alert message with 'hello'
- The string 'hello' being added to the end of the document's/page's HTML.

"Hello World!"

Let's put together the "Hello World!" code from above on a sample Web page. The page calls the function once when the page is loaded, and 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>
```

Extended "Hello World!"

In the following example, the function hello does "understand" whether it is called with a parametre or not, and returns the greeting with this parametre, or with the word "World":

```
<!DOCTYPE html>
<html>
  <head><title>Extended Hello World!</title></head>
  <body>
    <button id="msg">greeting</button>
    <script type="text/javascript">
      function hello(who) {
        if ((who == null)
         || (who.toString().search("object") >= 0)) {

          who = "World";
        }
        alert("Hello " + who + "!");
      }

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

      hello("guy");
    </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.

JavaScript/Anonymous functions

An anonymous function is a function that was declared without any named identifier to refer to it. As such, an anonymous function is usually not accessible after its initial creation.

Normal function definition:

```
function hello() {
  alert('Hello world');
}
hello();
```

Anonymous function definition:

```
var anon = function() {
  alert('I am anonymous');
}
anon();
```

One common use for anonymous functions is as arguments to other functions. Another common use is as a closure, for which see also the Closures chapter.

Use as an **argument to other functions**:

```
setTimeout(function() {
  alert('hello');
}, 1000);
```

Above, the anonymous function is passed to setTimeout, which will execute the function in 1000 milliseconds.

Use as a **closure**:

```
(function() {
  alert('foo');
})();
```

Breakdown of the above anonymous statements:

- The surrounding parentheses are a wrapper for the anonymous function
- The trailing parentheses initiate a call to the function and can contain arguments

Another way to write the previous example and get the same result:

```
(function(message) {
  alert(message);
}('foo'));
```

An alternative representation of the above places the initiating braces to the surrounding braces and not the function itself, which causes confusion over why the surrounding braces are needed in the first place.

```
(function() {
  // …
})();
```

Some have even resorted to giving the trailing braces technique derogatory names, in an effort to encourage people to move them back inside of the surrounding braces to where they initiate the function, instead of the surrounding braces.

An anonymous function can refer to itself via **arguments.callee** local variable, useful for **recursive** anonymous functions:

```
// returns the factorial of 10.
alert((function(n) {
  return !(n > 1)
    ? 1
    : arguments.callee(n - 1) * n;
})(10));
```

However, arguments.callee is deprecated in ECMAScript 5 Strict. The issues with arguments.callee are that it makes it impossible to achieve tail recursion (a future plan for JavaScript), and results in a different `this` value. Instead of using arguments.callee, you can use **named function expression** instead:

```
// returns the factorial of 10.
alert( (function factorial(n) {
  return (n <= 1)
    ? 1
    : factorial(n - 1) * n;
})(10) );
```

JavaScript/Object-based programming

Object-Based Programming

JavaScript supports *object-based* program design. (It also supports imperative and functional program designs.[1])

The purpose of an object-based design is to allow the components to be as modular as possible. In particular, when a new object type is created, it is expected that it should work without problem when placed in a different environment or new programming 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 isn't something you need to concern yourself with.

In contrast to object-oriented programming that uses classes, JavaScript does not have any; it uses objects to represent complex data types. Some objects are created manually, while others are built-in to the language itself, like the `Date` object. Objects are small structures of data with their own fields and functions to access or modify these fields. A benefit of objects is that they use a "black box" approach where the internal variables are protected from most forms of outside interference. This helps to prevent them from being altered by program code *outside* of the function or object, which provides a better integrity for the data itself.

Unlike other programming languages, JavaScript does not provide implicit protection levels on members of an object. In particular, JavaScript uses a prototypical form of objects, which can still inherit from parent classes. Although JavaScript is only an object-based and not an object-oriented language, most design patterns can still apply within the language as long as there is no attempt to directly access the object's internal state (for example, using an object's methods.)

As with many other programming languages, objects in JavaScript have their fields and functions referenced by the dot (`.`) between the object and field name.

The philosophy of object-based 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.

In JavaScript, you can make your own objects (like the *document* object) and even define your own variable types. Although JavaScript allows the creation of objects, access protection is not directly available; as such, it is possible to bypass this intent 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/Objects

Contents

Objects

An **object** is a code 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.

JavaScript provides a set of predefined objects. For example: an HTML document is itself (represented as) an `Object`, with internal variables like 'title' and 'URL' and methods like 'getElementsByClassName()'.

Basic example

```
var site = new Object(); //Required to not cause error in Internet Explorer
site = {};
site.test = function(string) {
        alert("Hello World! " + string);
        site.string = string;
}
site.test("Boo!");
alert(site.string);
```

The example would first

- define the site as an object
- define the site as a blank object

- define the site.test function
- call the site.test function with variable "Boo!"

The result:

- An alert message with 'Hello World! Boo!' will appear.
- site.string will be defined as string
- An alert message with 'Boo!' will appear.

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));
```

```
2018
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());
```

2018
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!".

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.

The `new` operator

The `new` operator creates a new instance of an object.

```
item = new Object();
```

Object methods and fields

In JavaScript, objects are free form - they can be modified at run time to instantly create a new object or to create new fields or functions.

```
money = new Object();
money.quarters = 10;
```

Object literals

Objects can also be created using the object literal notation.

```
money = {
    'quarters': 10
};
```

JavaScript/Constructors and prototypes

Constructor

The `new` operator creates a new object based on a constructor and a prototype.

The constructor is a function matching the name of the object, and, when called by the new operator, has the keyword `this` assigned to the newly created instance of the object. The constructor can then assign any initial variables as required.

```
function CoinObject() {
  this.value = 0;
}
var slug = new CoinObject();  // Creates a "slug" - a valueless coin
(slug.value = 0).
```

The constructor can accept parameters - however, it is not possible to overload the constructors by having multiple functions with a different number of parameters.

(The '`new`' statement is the most common way to create a new JavaScript object from a constructor function, but a few JavaScript programmers sometimes use alternative techniques.[1][2])

Prototypes

A prototype for an object is the set of auto-created fields and methods. It cannot operate by itself, and relies on an existing constructor.

When the object is created, the fields initialized in the prototype are copied to the newly created object.

```
function CoinObject() {
  this.value = 0;
}
CoinObject.prototype.diameter = 1;
slug = new CoinObject();  // A valueless coin (slug.value = 0), with diameter
of 1 (slug.diameter = 1)
```

Since the prototype of an object behaves as an object, you can use this to create inheritance.

Dynamically extending objects

Objects within JavaScript are fully dynamic, and fields within an object can instantly be created if they do not already exist. If, for example, you discover that you now need to keep track of the thickness in the coin example, you can simply add the new field to the prototype.

Changing the prototype of an object allows existing instances of the object to gain access to that change.

When a property value has been assigned on a new object, that property takes precedence over the prototype property from its ancestor. If you delete the property from the new object, the chain of inheritance takes you back to the property from the parent object.

To demonstrate, an `Animal` constructor is used to create an object, a dog called Spot. After that creation, the constructor has a gender property added to it. That gender property is also accessible from the object for Spot.

The gender property can be updated, and if the gender property is deleted from the dog object, it will retrieve the desired property from its ancestor instead, the `Animal` constructor.

```
function Animal (type, name) {
  this.type = type || 'No type';
  this.name = name || 'No name';
}
var dog = new Animal('dog', 'Spot');

// Add gender to the Animal object
Animal.prototype.gender = 'unspecified';
// dog.gender is 'unspecified'

dog.gender = 'male';
// dog.gender is 'male';

delete(dog.gender);
// dog.gender is once again, 'unspecified'
```

JavaScript/Inheritance

Contents

`instanceof` operator

The `instanceof` operator determines whether an object was instantiated as a child of another object, returning `true` if this was the case. `instanceof` is a binary infix operator whose left operand is an object and whose right operand is an object type. It returns `true`, if the left operand is of the type specified by the right operand. It differs from the `.constructor` property in that it "walks up the prototype chain". If object `a` is of type `b`, and `b` is an extension of `c`, then `a instanceof b` and `a instanceof c` both return `true`, whereas `a.constructor === b` returns `true`, while `a.constructor === c` returns `false`.

Inheritance by prototypes

The prototype of an object can be used to create fields and methods for an object. This prototype can be used for inheritance by assigning a new instance of the superclass to the prototype.[1]

```
function CoinObject() {
  this.value = 0;
  this.diameter = 1;
}

function Penny() {
  this.value = 1;
}
Penny.prototype = new CoinObject();

function Nickel() {
  this.value = 5;
}
Nickel.prototype = new CoinObject();
```

JavaScript/Access control

Access Control

JavaScript does not provide a direct means to control access to internal variables, however, it is possible to restrict access to some variables.

By default, variables within an object are public and can be modified anywhere in the code. As such, any programmer that uses the code in the future may change the internal state of the object unexpectedly, which has the potential for problems. While the easiest protection against this is to properly document your code (e.g. comments showing how to use the object), there are cases where you want to block direct access to a variable.

To declare and use a variable as private, there are two steps required:

- Declare a new variable within the constructor using the `var` statement.
- Create an anonymous function within the constructor, and assign it as a method for an object.

The example below shows a private field being used:

```
function MyObject() {
  this.publicNumber = 10;  // Public field.
  var privateNumber = 20;  // Private variable.

  this.getPrivateNumber = function() {
    return privateNumber;
  }
}

testObject = new MyObject();
```

`var privateNumber` is normally a local variable that only exists within the function. As you can see, it is accessed in `this.getPrivateNumber()`, which is an anonymous function, attempting to access it directly would result in an error. Since this anonymous function is declared in the constructor, it can use and modify the local variables declared in `function MyObject`, and keeps a reference to the variable even when the function returns. The anonymous function is bound to an instance of the object, and creating a new MyObject will create a new anonymous function that refers to the new `privateNumber`.

JavaScript/Closures

Contents

JavaScript closures

In programming languages having first-class functions, **closures** (that are also called **function closures** or **lexical closures**) are a technique for implementing lexically-scoped name binding. Concerning its operation, a closure is nothing but a data structure storing a function together with an environment, which maps all associating each free variable of the function (that are used locally, but defined in an enclosing scope) with the value or storage location the name was bound to *at the time the closure was created*. A closure allows the function to access those captured variables through the closure's reference to them, even when the function is invoked outside its scope.

More technically, a closure is a stack frame of a function that is not deallocated when it returns.[1]

Examples

```
function sayHello(name) {
    var text = 'Hello, ' + name + '!';
    var sayAlert = function() { alert(text); }

    return sayAlert;
}
var say = sayHello('John');

say();
alert(say.toString());    // Returns the code of the anonymous function
```

The function reference variable `say` references to both the function that alerts *and* to its closure! The closure is created by JavaScript, as the anonymous function is created *inside* another function. As the anonymous function needs its variables and those "inherited" from its enclosing function, the closure is necessary to provide it with these data. It is also possible to visualise the code of the function (see the last alert statement).

What makes it difficult to understand to many people is that in JavaScript, a function reference has also a hidden reference to the closure it was created in. This is similar to delegates that are a method pointer plus a secret reference to an object.

JavaScript/Debugging

Contents

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 [1]

- 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[2][3][4]. 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.

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.

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];
}
```

JavaScript/Strict mode

The strict mode

Strict mode can be enabled by placing '"use strict";' at the beginning of a script, before other statements:

```
// Dummy comment
"use strict";
var myvar = 4;
```

It can also be enabled for a single function only:

```
function myfun(){
  "use strict";
  var myvar = 6;
}
```

Strict mode ensures the following:

- New variables need to be declared with "`var`"; "`var`" is no longer optional.
- Attempts to write to non-writable variables throw an error rather than silently doing nothing.
- Attempts to delete undeletable properties throw an error rather than silently doing nothing.
- Octal numerals are disallowed.
- Etc.

Strict mode is available since JavaScript 1.8.5, i.e. ECMAScript version 5.