Introduction to JavaScript, Part 1

Luka Abrus
Technology Specialist, Microsoft Croatia

JavaScript is the scripting language used on millions of Web pages. Its main purpose is to add interactivity to the browser and Web pages. It also complements very popular server-side programming languages and platforms, like ASP.NET (actually, JavaScript is used in many ASP.NET controls). As it is very easy to learn, you'll soon start writing your first scripts. It doesn't matter if you're already an expert or just a beginner in the world of web development – JavaScript will add that "extra something" to your Web pages.

This guide assumes you have some prior knowledge of HTML and how web pages work.

What is JavaScript?

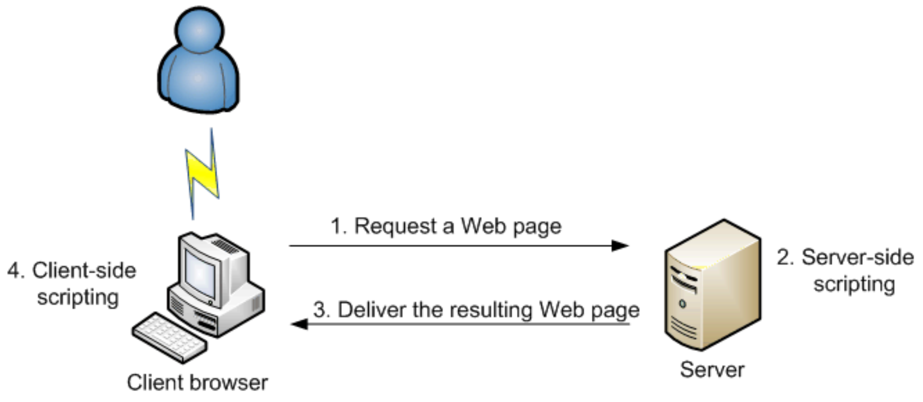
To understand JavaScript, you first need to understand what client-side scripting is. After you have mastered HTML and CSS, it is time to move one step forward. As you've seen up until now, the biggest limitation of HTML is that it is *static*. The page you've created will be shown to all visitors in the same way. For example, what if you wanted to show the current date on the Web page? Would you update its code each morning (or, better, after midnight) and upload it to the server? What if you came up with an even crazier idea of always having the current time on the page? Would you update your HTML every second?

Luckily, you aren't limited to only HTML. There are lots of different technologies that are just waiting to be used on your pages to create dynamic content! In this guide, we'll focus specifically on JavaScript. But before we move on, one more thing – what is actually a script? So far, all of your pages were static and written in HTML. By adding scripts, you are actually adding programming code to your pages. This code will run when the page loads, and it will add new functionality, interactivity and dynamic effects, unlike the HTML code used only for describing and formatting the page.

Client and server

Who is hiding behind the word "client" when talking about Web technologies? When you request a Web page by entering its address in the browser's address bar, you and your browser are becoming a *client*. As in the real world, you have requested a service and therefore you are a client. On the client-side, as you'll see, your browser does all the work of processing the service and displaying the page.

JavaScript is a client-side scripting language. In order to understand what that means, you need to understand what the terms "server-side" and "client-side" mean. Let's look at how the process of requesting a Web page goes.



After you have entered the address of the page you want or clicked a link, your browser requests that page from the server (1). The server then finds the page, opens it and runs all server-side scripts in it. For example, if it is an ASPX (ASP.NET) page, the server will execute all code on the page (2). After all the server-side scripts are processed, the result is sent back to the browser (3). The result is a page that has HTML and CSS code, and optionally, client-side scripting code. If there is client-side scripting code on the page (for example, JavaScript), the browser will process it (4) and display it to the user. It will also keep processing it and running it when necessary as long the page is displayed in the browser.

Now let's dive in – server-side scripting (like ASP.NET, ASP or PHP) is used for generating Web pages by request. All data on such pages will be current, as they are created moments before being sent back to the browser. Examples are online shops, auction sites, online forums and bulletin boards. It is important to note that, although there is a lot of programming code being run server-side on these pages, they are given to the browser as pure static HTML. The user has no idea how the page was created or that it is a dynamic page. All the scripting happens, as its name states, on the server.

On the other hand, client-side scripting is used to make pages interactive after they are sent to the browser. A common usage of client-side scripting is to check the data that the user has entered in a form on the page. For example, if the user forgot to enter his full name or misspelled the e-mail address, the client-side script will warn the user about such errors. Client-side scripts are also much more responsive and faster for the user, unlike the server-side scripts which depend on the server and Internet traffic between the user's computer and the server, as the request has to be sent to the server and then back again.

Look at some actions you might want to do on your Web page and the proper technology to do it:

Action	**Use…**
Disable right mouse click	Client-side scripting
Show data from the database	Server-side scripting
Secure your page (login form)	Server-side scripting
Visual page effects (e.g. dynamically swap images on mouse over)	Client-side scripting

Show information in new browser window	Client-side scripting

So in this guide, we'll focus on client-side scripting with JavaScript. Although JavaScript is the most popular client-side scripting language, it is not the only one. You could also create scripts in VBScript, but we suggest you to stick with JavaScript as it is understood by all browsers.

Also, a word of caution – don't confuse JavaScript with the Java programming language. They have nothing in common (except the word "Java" in their name). JavaScript is a scripting language, which means the browser reads it and interprets it directly, while Java code needs to be compiled. Java applets are Java web components, but they don't have anything to do with JavaScript. So there is no such thing as JavaScript applets or Java script, just to make it clear.

To wrap up with this introduction, let's repeat the key points:

- Client-side scripting represents all code that is executed in the browser, after the server sends the page back to the user.
- Client-side scripting is completely independent of the server and server-side technologies, as the browser couldn't care less if the page was produced with ASP.NET on Windows or PHP on Linux, just as long it receives the HTML code it can display and the client-side scripts it can process.

JavaScript was first introduced in 1995 and here lies the key to its popularity. Back then there were no server-side scripting technologies, and if you wanted some interactivity on your web pages, JavaScript was your only choice. Now, more than a decade later and after dozens of server-side technologies have been introduced, JavaScript is experiencing a revival through AJAX scripts. We won't get much into that here, but just to let you know – JavaScript is the core engine of every AJAX script and if you learn JavaScript, then you're on your way to learning how to create AJAX-enabled pages.

Your first JavaScript script

Let's start programming JavaScript scripts! In this guide we'll show you how to use JavaScript, and even more importantly, teach you when and for what purpose it is best to use JavaScript.

As opposed to server-side scripts, which are executed the moment a user requests a page, client-side scripts don't necessarily have to be executed all at once. Client-side scripts can, of course, be run right after loading the page in the browser, but they can also be run as a response to a user's action; for example, when a user clicks on a button or when they move the mouse over an element.

All JavaScript code must be put within a special HTML tag, the **SCRIPT** tag.

```
<html>

<head>

    <title>My page</title>

    <script type="text/javascript" language="javascript">

    <!--
    // Your JavaScript code


    //-->
    </script>
</head>
<body>


</body>
</html>
```

All JavaScript code should be put inside of the **SCRIPT** element. We've put our **SCRIPT** element inside of the page header (**HEAD**), but that is not mandatory – you can put a **SCRIPT** element anywhere on the page. And more importantly, you can have as many **SCRIPT** elements as you like!

In our example we've also put comment signs inside of the **SCRIPT** element (<!--and //-->). This is for backward compatibility with older browsers that can't understand JavaScript code. Although probably 99% percent of today's browsers can process JavaScript code and will ignore the comment signs, you should still use them, just in case. We'll omit these comment signs in future code listings for brevity sake.

We've also used JavaScript comment signs - "//". Lines beginning with two forward slashes won't be processed and you can use them to write anything, comments, instructions, your to-do list, etc.

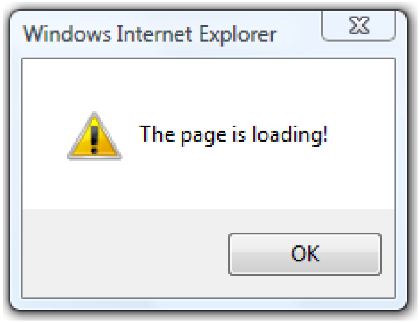
Let's go on with writing some code. Most of your JavaScript code will be written as functions. Functions are blocks of code you can call and which can return a certain value or do a certain action. You can call functions as many times as you want and use parameters when calling them, which can result in different actions. Also, functions can be called as a response to any action in the browser, like the mouse click.

The JavaScript code that you write outside of a function (but still within the **SCRIPT** element) will be run when the page loads. The browser reads the page and its HTML code from top to bottom. If it reads a normal HTML tag, it will display it. Same with JavaScript – if it reads a JavaScript command, it will execute it. But if it reads JavaScript function, it will store it in memory and execute it later when needed.

Here's an example:

```
<html>
<head>
    <title>My page</title>
    <script type="text/javascript" language="javascript">
    <!--


    function say(text)
    {
        alert(text);
    }

    say("The page is loading!");

    //-->
    </script>
</head>
<body>
<noscript>
JavaScript is turned off.
</noscript>

<input type="button" value="Click me!" onclick="say('Thanks for
clicking!')" />

</body>
</html>
```

This example features a simple function. It takes a parameter and then displays it to the user using *alert* method. The browser will read the page, read the function and store it in memory and then read the JavaScript line that calls the function with parameter "The page is loading!". This line will call the function and show the text. All of this is happening while the page is loading, and before the browser has read the rest of the page. Here's the result of this call:



The page is loading!

After showing this message, the browser continues with reading the page. We've used a **NOSCRIPT** element in the body of the page which is displayed only if JavaScript is disabled or unavailable in the browser. Some users (to be more precise, around 4% of them, as statistics show) intentionally turn JavaScript off and therefore JavaScript scripts won't run in their browsers. Inside of a **NOSCRIPT** element you can write a message like "To fully experience this page, please turn on JavaScript" or something similar.

The page also contains a simple button, but with a special attribute called *onclick*. This attribute contains JavaScript code that will be run when the button is clicked (which is rather logical, if you look at the attribute name). So when the user clicks on the button, it will call our JavaScript function *say* with the text "Thanks for clicking!".

Pay attention to the quotation marks used – the *onclick* attribute has its value inside of double quotation marks. But to call the function *say* with a text parameter, we also must use quotation marks, so we use single quotes to distinguish them from the *onclick* attribute quotes.

Try this page out – while loading, the browser will display the text "The page is loading!" and then load the rest of the page and display it. And then it will stop and wait. Nothing will happen until you click the button. You can click the button as many times you like, it will always display the message "Thanks for clicking!".

Open up Visual Web Developer Express Edition, select File | New File | HTML Page and copy the JavaScript above into this page.

You can also put another **SCRIPT** element at the end of the page and use it to call the function *say* defined earlier on the page.

```
<script type="text/javascript" language="javascript">

say("The page has almost loaded.");

</script>
```

You can also put all your JavaScript code in an external script. You can save all the code without a **SCRIPT** element (just as pure JavaScript code) in a special file and then link it to the page using **SCRIPT** element and its *src* attribute. For example, if you've put JavaScript code in file called "myscript.js", you could write:

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

</script>
```

This way your HTML page looks much clearer, and if you follow the practice of keeping your CSS file separate from the HTML file, you can also do this with JavaScript code. In addition, you can link this external JavaScript file to multiple, different pages. If you create a function you want to use on several different pages, put it in an external JavaScript file and link it to all those pages.

One of the downsides of client-side scripting is that all your visitors have access to the source code of your scripts. With a simple *View – Source* in Internet Explorer, users can access the complete source code of the page. As this source code also contains client-side scripts (for the browser to execute), it's easy to copy them. But when beginning programming in JavaScript, that can be an advantage. If you spot some interesting functionality on a page that you want to use on your own page, nothing is stopping you from looking at the source code and analyzing the script.

We'll look into functions, events and other topics shown so far with more details later in this guide. Let's start now with JavaScript basics.

Quiz 1

Question 1.1: Here are a few programming actions that are usually run on the server, on the client or can be run on both. Check the appropriate boxes.

Action	Server	Client
Online browser games		
Online questionnaires with results		
Validation of user’s input in forms		
Scrolling text		

Answer 1.1: Online browser games are usually done with Flash, which is run on the client, in the browser, as they need to be very responsive.

Online questionnaires are done using server technology, as the results need to be stored in a database.

Validation of user’s input in forms is usually done both on the server and on the client. The client in this case does the validation to save user’s time if the form isn’t filled correctly and displays errors and instructions immediately. The server does the final checking before any action is done and before user input is processed.

Scrolling text is a typical client job, as the script that runs in the browser displays and updates the text, thus creating a scrolling effect.

Action	Server	Client
Online browser games		X
Online questionnaires with results	X	
Validation of user’s input in forms	X	X
Scrolling text		X

Question 1.2: How many SCRIPT elements with JavaScript code can you have on a page?

Answer 1.2: As much as you like! The SCRIPT elements will be processed in the order they appear on the page. Usually (but not necessarily) only one SCRIPT element with the whole script is used and is put inside of HEAD element.

Question 1.3: Can JavaScript code run in all browsers?

Answer 1.3: No. Although now all browsers support JavaScript, some users don’t actually like JavaScript and they intentionally turn it off (statistics show that around 4% of users don’t want or can’t use JavaScript). In that case nothing happens, JavaScript code simply isn’t run and the page stands still. You can use a NOSCRIPT element to display a specific message to all users not running JavaScript.

JavaScript basics

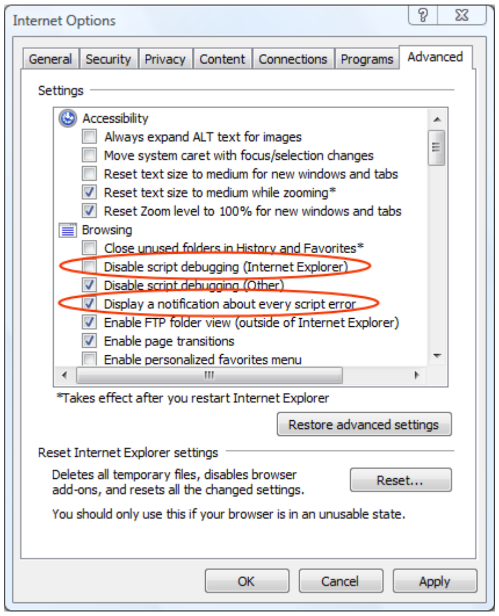
The first thing you need to remember when programming is that JavaScript is case-sensitive, meaning that there is a big difference between lower and upper case letters.

So far you've learned how to call the *alert* method, which displays the small windows with a message. None of the following lines would have the same result:

```
Alert("Hello!");

ALERT("Hello!");

aLeRt("Hello!");
```

All three methods (Alert, ALERT and aLeRt) are unknown to JavaScript, so it expects them to be one of your specific functions you created in the code, like the *say* function in the previous example.

When writing JavaScript code, it is very useful to turn debugging on in your browser and to enable displaying all JavaScript errors. Otherwise you won't know when something is wrong with your code and you'll have no idea where the errors are. In Internet Explorer click on Tools – Internet Options and then on the Advanced tab.


Internet Options

General | Security | Privacy | Content | Connections | Programs | Advanced

Settings

Accessibility
- ☐ Always expand ALT text for images
- ☐ Move system caret with focus/selection changes
- ☐ Reset text size to medium for new windows and tabs
- ☑ Reset text size to medium while zooming*
- ☑ Reset Zoom level to 100% for new windows and tabs

Browsing
- ☐ Close unused folders in History and Favorites*
- ☐ Disable script debugging (Internet Explorer)
- ☑ Disable script debugging (Other)
- ☑ Display a notification about every script error
- ☑ Enable FTP folder view (outside of Internet Explorer)
- ☑ Enable page transitions
- ☐ Enable personalized favorites menu

*Takes effect after you restart Internet Explorer

Restore advanced settings

Reset Internet Explorer settings

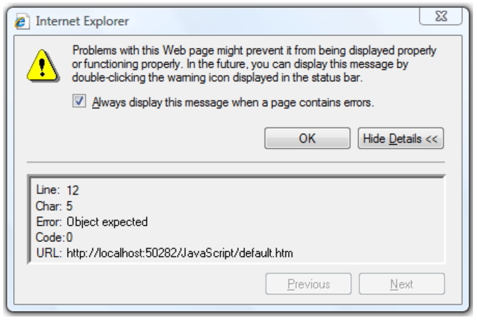
Deletes all temporary files, disables browser add-ons, and resets all the changed settings.

Reset...

You should only use this if your browser is in an unusable state.

OK | Cancel | Apply

Make sure you uncheck the option "Disable script debugging (Internet Explorer)" and check the option "Display a notification about every script error". After clicking OK and opening the page with, for example, an Alert call like in the previous example, Internet Explorer would show the following error:


Internet Explorer

Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.

☑ Always display this message when a page contains errors.

OK | Hide Details <<

```
Line: 12
Char: 5
Error: Object expected
Code: 0
URL: http://localhost:50282/JavaScript/default.htm
```

Previous | Next

This way you can even track the line of code which caused the error.

To work with data in JavaScript, as in every other programming or scripting language, you will use variables. JavaScript doesn't care about variable types, which means you can easily mix text with numbers, as we will show in the following example. Also, you don't necessarily have to declare variables, you can just start writing their values (we'll cover this in more detail later, when we get to functions and local and global variables). Look at this example:

```
<script type="text/javascript" language="javascript">
/*
Example: Variables
*/
a = "Hello world!";
b = 5;
c = "17";
d = 3.25;
alert(a + b);  // "Hello world!" + 5 = "Hello world!!5"
alert(b + c);  // 5 + "17" = "517"
alert(b + d);  // 5 + 3.25 = 8.25
</script>
```

Similar to the C programming language, JavaScript uses two forward slashes to specify comments in the code (anything after two forwards slashes on the line won't be processed and will be ignored) and also "/*" and "*/" signs (anything between those signs, no matter how many lines that includes, will be ignored).

In our example, variables *a* and *c* contain text, while *b* and *d* contain numbers. The plus sign ("+") has two meanings – it can be used for concatenation (combining two strings) or addition (if it's applied to numbers). In the comments you can see the results of the calls. If you combine text with a number, the result will always be text. Only combining a number with a number results in a number (like *b* and *d*).

When working with variables, just like with methods, pay attention to capitalization. Variable *A* isn't the same as *a*, nor is *B* the same as *b*.

Although *c* may look like a number to us, it is actually a string, because it is surrounded by quotation marks. Remember our first call to the *alert* method where we supplied the text to be displayed? It was written inside of quotation marks, because it was text. You can use either double or single quotes, but be sure not to mix them.

```
alert('Hello!'); // Correct!
alert("Hello!"); // Correct!
alert('Hello!"); // Incorrect!
```

Let's just mention semi-colons at the end of lines. In JavaScript they aren't mandatory, but if you have programmed in C before, you may be more comfortable with this format. There is no reason to use semi-colons, but it can help make your code more readable, and it won't hurt the functionality or the speed of JavaScript.

Functions

You've already seen a function in our first example. We've created a function called *say* that displays an alert and then called it from different places in our code. Generally speaking, functions are used for containing and executing blocks of code. Once you define and create a function, you can call it as many times as you like. If you put it in a separate JavaScript file, then you can even call it from different pages.

Functions are defined with the **function** keyword followed by the name and parentheses. Inside the parentheses you can, optionally, define any variables that are given to the function – these variables are called parameters of the function. Your functions can take as many parameters as you need (of course, not exactly, since there is a limit, but you'll hardly reach it in your programming tasks). Function code must be placed inside curly braces. Look at the following examples:

```
<script type="text/javascript" language="javascript">
function sayHello() {
    alert("Hello!");
}

function say(text) {
    alert(text);
}

function addNumbers(a, b) {
    result = a + b;
    return result;
}

sayHello();
say("How are you doing?");

x = addNumbers(5, 17);
alert(x);
</script>
```

We have defined three functions which are rather different. The first one is called *sayHello* and it doesn't take any parameters, as you can see by its empty parentheses. We call this function from the code by its name:

```
sayHello();
```

The second function called *say* takes one parameter. Inside the parentheses in the function definition we define the parameter name. With this name in the function code we can access the value that is supplied to the function when called. In the case of the function *say*, the action is simple – we just display the value supplied to the function using the *alert* method. Function *say* is much more flexible then the *sayHello* function, as it can take any parameter and display it to the user, for example:

```
say("How are you doing?");

say("How are " + "you doing?");

say("I'm great.");
```

As opposed to the function *say* which takes only one parameter, the function *addNumbers* takes two parameters. In the function definition they are separated with a comma, and the first variable name (*a*) will be used for the first parameter given to the function, and the second (*b*) for the second parameter.

There is one more difference – function *addNumbers* does something and then returns the value. Function *say* and *sayHello* were independent of the rest of the code, as they would just do their job when called and display the text. Function *addNumbers* is different as it adds two numbers and then returns the result back to the main line of code that called it.

```
x = addNumbers(5, 17);

alert(x);
```

Variable *x* is used to store the result of the function *addNumbers*. It is called with two parameters that have values 5 and 17 (in the function code they will be accessed through variables *a* and *b*). The important thing to note in the function is the **return** command – with the **return** keyword we return the result of addition of two parameters. That value is stored in variable *x* in the main code and displayed with the *alert* method.

Local and global variables

Functions are independent parts of code, so they can create and use variables that already exist in the main code (or the code outside of any function). Such variables are called *local variables*. To declare a local variable inside of the function use the *var* keyword. Local variables are totally independent of the rest of the JavaScript code, as they are used only inside the function that they are declared in. For example:

```
number = 5;


function showNumber() {

   var number = 12;

   alert(number);

}


alert(number);

showNumber();

alert(number);
```

The result of this example would show the messages “5”, “12” and then “5” again. The variable *number* which is declared with the value 5 outside of any function is a global variable. The code first displays the global variable *number*. Then it calls the function *showNumber* which has its own local variable called *number* with value of 12. After displaying "12", the function ends and we go back to the main code. There we display again the global *number* variable with value of 5.

Functions can normally return only one value with the **return** command. So if you want to return more than one value, you will have to use global variables. Every variable that isn't explicitly declared with **var** in the function will be considered a global variable.

```
number = 5;


function addNumbers(a, b) {
    number = a + b;
    alert(number);
}


alert(number);
addNumbers(2, 9);
alert(number);
```

This example would display themessages "5", "11" and "11". We first display the global variable *number* that has value of 5. Then we call the function *addNumbers* with parameters 2 and 9. Inside the *addNumbers* function we use the *number* variable without the **var** keyword. Therefore, the global variable *number* is used. After displaying the result ("11"), we return to the main code where the *number* variable has also changed (as it is the same variable as in the function). Then, just to make sure the variable has changed in the function, we display its value (still "11").

Local and global variables can give you headaches if not used properly. Sometimes you'll forget you've already used a certain variable name in the main code and then reuse it in the function. That will result in changing the global variable, which might lead to unexpected behavior of your script (and to your wondering where that value came from).

One other reason why we explained local and global variables (aside from being able to return multiple values from a function, and saving your time when you get mixed up in using the same variable name multiple times) is to show you how the code is actually executed. It is read line by line and the functions are executed only when called. Functions can call other functions, but if you put all of your code inside functions and nothing as the main code, then nothing will happen.

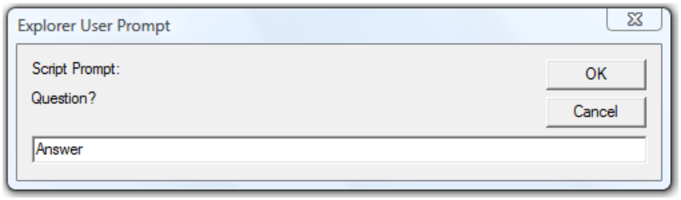
If clause

All the code you've written so far has been a bit simplistic – all the commands have been executed consecutively. The browser reads your code line by line and executes your commands accordingly (of course, except functions which would be executed only when called). That doesn't necessarily have to be the case, and, as you'll see when you get to more programming, you'll almost never want to run your code completely from the beginning to the end without exceptions.

We'll show you a simple example of this – say you want your user to enter his year of birth. We've already seen how to display information in a window to the user (using the *alert* method). Now, to prompt the user to enter information we'll use the *prompt* method. This method can have two parameters, one with the question and one with the default answer. Unlike the *alert* method, the *prompt* method returns a result - the text user has entered in the window.

```
result = prompt("Question?", "Answer");
```

This would result in the following window being displayed.



So, let's say you wanted to compare the user's year of birth with your own (for example, 1986) to check if you're the same age. If his year of birth is the same as yours, you would display a message about it. Here's the code to do it:

```
<script type="text/javascript" language="javascript">
yearOfBirth = prompt("Your year of birth?", "1986");
if (yearOfBirth == 1986) {
    alert("You were born the same year as me!");
}
</script>
```

After prompting the user we use the **if** clause to check whether a certain condition is true. If the condition inside parentheses is true, then all of the code within the curly braces will be run. In this case it will display the message saying "You were born the same year as me!". If not, the script would just skip that part and go on as if nothing happened.

Here are few operators you can use to check conditions:

Operator	**Description**
<	Less
<=	Less or equal
==	Equal
>=	Bigger or equal
>	Bigger
!=	Not equal (different)

All of these operators result in true or false values. For example, here's how you could check several conditions:

```
number = 15;
```

```
if (number > 10) { } // true
if (number != 39) { } // true
if (number <= 15) { } // true
if (number == 10) { } // false
```

Be very careful when checking if a variable is equal to a certain value. Be sure not to mix up the "=" and "==" signs. The first one ("=") would set the variable to the value you wanted to compare it with, and the other one would compare it to the value and return true or false. So if you wrote the following code by mistake, the **if** clause would always be true and all of the code within **if** clause would be run:

```
number = 15;
if (number = 30) {
    alert("The number is now 30!");
}
```

Let's get back to our example with user's year of birth. The **if** clause can have a more complex form. Here's how you would show a proper message if the user is not the same age as you.

```
if (yearOfBirth == 1986) {
    alert("You were born the same year as me!");
} else {
    alert("Nevermind.");
}
```

The script first checks the condition in the **if** clause, and if it's true, it runs the code within the curly braces of the **if** statement. If it's not true, for example *yearOfBirth* is 1983, the script continues with the **else** statement and runs that code.

If the first condition is met, after running the **if** statement code, the script skips over the **else** statement and continues with the script after the whole **if** clause. This way the script makes a decision which block of code to execute. The script will never display both of the messages, as either *yearOfBirth* is equal to 1986 or it simply isn't.

You can also combine multiple **if** statements, for example to check if the user is older or younger than you:

```
myYearOfBirth = 1986;
yearOfBirth = prompt("Your year of birth?", "1986");

if (yearOfBirth == myYearOfBirth) {
    alert("You were born the same year as me!");
} else if (yearOfBirth < myYearOfBirth) {
    alert("Hey, you're older than me!");
} else {
    alert("Did you know you were younger than me?");
}
```

This code is completely readable. We first check if user's year of birth is the same as ours (defined previously in the code to be 1986). If this first condition is not true, the code continues with the next *else* statement in which we check if the user is older than us. If that condition is not met either, the code goes on to the final *else* statement and executes all of that code. If, for example, the first condition is met and *yearOfBirth* is equal to *myYearOfBirth*, then the code displays the message "You were born the same year as me" and finishes with the whole **if** statement without checking any other conditions.

What happens if the user, when prompted to enter his year of birth, doesn't click OK, but clicks Cancel? What happens with the variable *yearOfBirth* in the script code? The answer is simple – it becomes *null* (or to say it in a more friendly way, unknown or undefined). So how would you check if the user has clicked OK or pressed enter in the prompt?

JavaScript follows a simple, yet perhaps at first sight silly, rule – "true is everything that is not false". Or, to say it in a different way, JavaScript will consider any expression as truth as long it is not *null* or 0. This way you can easily check if a given object exists in JavaScript:

```
if (document.all) { ... }
```

If it does exist, the **if** clause would return a *not-null* value, which would be considered true and the script would continue with the code within curly braces.

In a similar way, we can expand our example:

```
myYearOfBirth = 1986;
yearOfBirth = prompt("Your year of birth?", "1986");

if (yearOfBirth) {
    if (yearOfBirth == myYearOfBirth) {
        alert("You were born the same year as me!");
    } else if (yearOfBirth < myYearOfBirth) {
        alert("Hey, you're older than me!");
    } else {
        alert("Did you know you were younger than me?");
    }
} else {
    alert("Why didn't you click OK?");
}
```

Can you still follow this code? We first check if *yearOfBirth* is true, or, as we explained, if it is defined and not null. If it is, we continue with the block of code within curly braces, which is familiar from previous examples. If it isn't, the script skips to the **else** statement and just alerts the user with the message "Why didn't you click OK?".

This programming style is very useful when you want to prepare for certain errors and be sure your code handles all possible execution paths. As we'll show later in this guide, error handling is crucial when writing scripts and it is actually completely based on **if** statements such as the one shown here.

If blocks can also simultaneously evaluate multiple different expressions. If you want one clause to check two or more conditions, you need to combine these conditions using logical operators. We'll show you how to use three of them – the AND operator ("&&"), the OR operator ("||") and the NOT operator ("!"). And just to help JavaScript decide which conditions to combine, use parentheses to separate them.

```
if ( (number >= 10) && (number <= 20) ) {
    alert("The number is between 10 and 20!");
}
if ( !(number == 10) ) {
    alert("The number isn't 10!");
}
if ( (number == 15) || (number == 16) ) {
    alert("The number is 15 or 16!");
}
```

The first one checks if the number is greater or equal to 10 and less or equal to 20. The second one checks if the number is not equal to 10 (you could also write this as "number != 10", but this way we first check if the number is equal to 10 and then perform a negation on the result). And the last one checks if the number is 15 or 16.

As you'll soon learn, **if** statements will be the foundation of most of your scripts. Let's go now to the next level – executing the same code multiple times.

Quiz 2

Question 2.1: What is wrong with the following code? After correcting it, what would be the message displayed to the user?

```
myText = "world";
alert("Hello ' + mytext);
```

Answer 2.1: There are two errors, both on the second line. First, appropriate quotation marks should be used (either only single quotes or only double quotes, but without mixing). Secondly, JavaScript is case-sensitive, so *myText* and *mytext* are two different variables. The corrected versions is shown below and the resulting message would be "Hello world".

```
myText = "world";
alert("Hello " + myText);
```

Question 2.2: What values would the variables *a*, *b* and *res* have (displayed to the user with an *alert* method at the end) after running the following code?

```
function math(x, y) {
    var a = x + y;
    b = x - y;
    return (a + b);
}
a = 10;
b = 15;
res = math(a, b);
alert("A: " + a + "; B: " + b + "; Res: " + res);
```

Answer 2.2: In the function *math*, variable *a* is a local variable (new and different from the one existing in the main part of the code), while the variable *b* is actually a global variable (the same one as in main part of the code). So variable *a* wouldn't change its value (it remains 10), variable *b* changes its value after the function call (and would be -5), while the variable *res* (result of the function call) equals 20.

Question 2.3: Write JavaScript code to take the user's input and if he enters "OK", display a simple message. But do it using only one line of code!

Answer 2.3: If you wanted to write this code the long way, you could write:

```
input = prompt("Give me your input:", "");
if (input == "OK") {
    alert("OK!");
}
```

But since the variable *input* would be used only one time, we can directly compare the result of the *prompt* method with value "OK". And when writing **if** statements, you don't need to use curly braces if there is only one command to be executed.

```
if (prompt("Give me your input:", "") == "OK") alert("OK!");
```

Looping through code

Loops are used for repeating commands a certain number of times, or until a certain condition is met. There are lots of real life examples of loops. In a grocery store when you want to buy, for example, 6 apples, you'll repeat the action of taking an apple and putting it into your basket exactly six times. And when you get to the cash register, you'll repeat the action of taking the money out of your wallet until you give an amount that is greater or equal to the amount on your bill.

For loop

If you have previously programmed in the C programming language or languages similar to it (like C++ or even C#), then you'll see that loops are pretty much programmed the same way in JavaScript. First you'll meet the **for** loop, which is used to execute a block of code for a specified number of times.

```
sum = 0;
for (i = 1; i <= 100; i++) {
    sum = sum + i;
}
alert("Sum of first 100 numbers is: " + sum);
```

This example adds the first 100 numbers together. A **for** loop is constructed out of three parts, each separated by a semicolon. The first part defines a variable for the counter and sets its initial state ("i = 1"). The second part defines the condition that has to be met in order to continue with execution of the code inside the loop. When the condition isn't met, the loop finishes and the code continues with the first next command, in this case, the *alert* method. The third part is the command to increment the counter. You can increment the counter by 1 (like in this case), or by any other number.

A side note – in JavaScript you can write additions and subtractions in a shorter way than "i = i + 1". For example, if you want to increment a number by 1, you can write either of the following commands:

```
number = number + 1;
number++;
```

Similar to operator "++", which increments a number by 1, you can use operator "--" which decrements a number for 1. So, again, both of the following commands do the same thing:

```
number = number - 1;
number--;
```

To continue with simplified ways of writing math operations, here are a few more. Take a look at the following four operations:

```
number = number + 10;
number = number - 39;
number = number * 3;
number = number / 15;
```

If you want to perform any of those operations on a number, here's how you can write them more easily.

```
number += 10;
number -= 39;
number *= 3;
number /= 15;
```

Instead of repeating the variable name on which you perform the operation after the equal sign, you can write the operation together with the equal sign and it will be applied to the variable. So, to go back to our example of the **for** loop, here's how you can add the value of the counter variable *i* to the variable *sum*:

```
for (i = 0; i <= 100; i++) {
    sum += i;
}
```

Explanation of this **for** loop is rather simple. The code within its curly braces will be executed exactly 100 times. The counter variable will be incremented after each execution of the code and it will be added to the variable *sum*. At the end, the variable *sum* will contain the sum of the first 100 numbers.

Do and While loops

But sometimes you won't know exactly how many times to repeat a part of code. In that case, you can choose between two different loops, a **do** loop and a **while** loop. Both of them are pretty similar and you'll be choosing based on your personal preference. Let's show you how they work.

For example, if you want to keep questioning the user until he gets the answer right, here's how you would do it with a **do** loop.

```
do {
    answer = prompt("What is the third letter of the alphabet?", "");
} while (answer != "c" && answer != "C")
alert("Thank you!");
```

The **do** loop keeps repeating the code inside the curly braces as long as the *while* condition is met. So, as long as the user doesn't answer "c" or "C", the loop continues and the user is prompted with the question. The main thing to note regarding **do** loops is that the code inside of the loop will be run before evaluating the *while* condition.

That is not the case with the **while** loop. The **while** loop first evaluates the code within the *while* condition and then runs the code inside the curly braces. But that is rather obvious when you look at the way it is written. Here's how you would calculate the sum of the first 100 numbers:

```
counter = 1;
sum = 0;
while (counter <= 100) {
    sum += counter;
    counter++;
}
alert("The sum of first 100 numbers is: " + sum);
```

First we initialize the variables *sum* (for calculating the sum) and *counter* (for counting first 100 numbers). As opposed to the **for** loop, in a **while** loop we have to manually increment the counter inside the loop code. If you forget to do that, then the counter will always be less than 100 and the loop will continue endlessly (or until you stop it by closing your browser).

Those are the three loops you can use in JavaScript. As you've seen, they work pretty much the same way – you can choose to work with the one you like the most, as all the actions can be accomplished by any of them.

Quiz 3

Question 3.1: What is the value of the variable *number* after running the following code?

```
number = 15;

number += 10;

number /= number;
```

Answer 3.1: It is 1. We first add 10 to *number* and then divide it by itself. We could also write this code the longer way:

```
a = 15;

a = a + 10;

a = a / a;
```

Question 3.2: How many times would the message "Hello!" be displayed to the user in the following **for** loop?

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

    alert("Hello!");

    i++;

}
```

Answer 3.2: Only 5 times. Although the **for** loop should run 10 times, we actually increment the counter variable once more in the code of the loop. Therefore, after each run, the counter would increment by 2 and the message would be displayed only 5 times.

Question 3.3: How many times would the message "Hello!" be displayed to the user in the following **while** loop?

```
while (true) {

    alert("Hello!");

}
```

Answer 3.3: It would be displayed endlessly, since the condition is always true (we're using the Boolean value *true*). Don't try this in your browser – you would have to kill the browser's process in the operating system to stop this loop. If you want to stop the loop while running, use the **break** command. For example:

```
while (true) {
    pwd = prompt("Enter password:", "");
    if (pwd == "password") break;
}
```

This loop would endlessly ask for the password and it would end only when the user enters the word "password". This kind of functionality (although maybe somewhat irritating to the user) can be achieved by putting the password check in the *while* condition of the **do** loop as well:

```
do {
    pwd = prompt("Enter password:", "");
} while (pwd != "password");
```

JavaScript objects

JavaScript supports working with objects. There are several useful objects already available in JavaScript, and you can create your own, but that will not be covered in this guide. Objects are used for working with specific data and have different functions which can help you in your programming tasks.

Important note – you might have heard about object oriented programming. Merely supporting objects does not imply that a language is an object oriented one. There are three different principles of object oriented programming: encapsulation, inheritance and polymorphism. This guide will not cover such advanced topics, but it is interesting to note that JavaScript actually is an object oriented programming language and all of those three principles are supported within the JavaScript language.

We'll explain two different objects available in JavaScript. There are many more of them, but now we'll focus on the two most useful objects for working with strings and the web page (or the *document*) itself.

String object

The String object, used to represent text, is the most common object in JavaScript. We've already used strings in our examples. So to repeat what you've already learned and add something new, here are two ways to define new strings.

```
var myText1 = "This is my text.";
var myText2 = new String("This is my text.");
alert(myText1 + " " + myText2);
```

We've used the *var* keyword for declaring new variables, but that is completely optional and is important only when working with local and global variables, as explained previously in this guide.

So, you can simply write the name of the variable and its value, like in the first example, or you can create a new object. With the operator *new* we create an object of type *String* and define its value as in the second line of code. Both the first and the second lines of code have the same effect, so we'll be using the first one in the next examples as it is simpler.

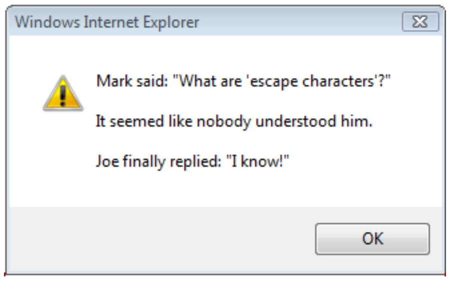
Within strings you can use special escape characters. Here's a table of the more useful ones:

Escape character	**Description**
\n	New line
\t	Tab
\"	Double quotation mark
\'	Single quotation mark

You'll be using escape characters for displaying information to the user. For example, if you want to display a more complex message box, here's how to do it:

```
alert("Mark said: \"What are 'escape characters'?\"\n\nIt seemed like
nobody understood him.\n\nJoe finally replied: \"I know!\"");
```

All escape characters are prefixed with a backslash sign. For example, if you put "\n" inside of the text, it will continue on the next line. This example would result with the following message box:



```
Mark said: "What are 'escape characters'?"

It seemed like nobody understood him.

Joe finally replied: "I know!"
```

When checking user's input, it's useful sometimes to convert text to upper case or lower case. When you want to question your user, it's complicated to anticipate all the possible answers. For example, if you ask him about the capital of France, the user might answer "Paris", "paris", "PARIS" or even something like "pArIS". In such case, the best thing is to convert the answer to upper or lower case and compare it with the right answer.

```
answer = prompt("What is the capital of France?", "");
if (answer.toUpperCase() == "PARIS") alert("Correct!");
```

To convert the text to upper or lower case, use *toUpperCase* or *toLowerCase* methods respectively.

```
myText = "Hello world.";
myUpperText = myText.toUpperCase(); // HELLO WORLD.
myLowerText = myText.toLowerCase(); // hello world.
```

These methods are applied directly to the variable *myText*. Just add a dot and the method name to the variable. If you want to convert the variable itself, don't forget to apply the new value to the variable itself using the "=" operator.

```
myText = "Hello world.";
myText = myText.toUpperCase();
```

The result of the previous example would be "HELLO WORLD.". As you see, these methods convert only the letters. Their usage is simple, but pay attention to the capitalization of the method names, since *touppercase* or *ToUpperCase* won't do anything and will result in an error.

Later in this guide we'll cover the most common usage of JavaScript on the web: validation of user input. In most cases you won't be using the *prompt* method to get input from the user, because HTML offers a more natural way to enter data – through forms. So, to be able to validate forms, you must learn a few more tricks with strings.

First of all, we'll show how to find something within a string. For example, if you want to validate an e-mail address, you must make sure it contains the characters "@" and "." (note that this is not enough for 100% validation of an e-mail address, but it will work for now). To check whether a string contains another string, use the *indexOf* method.

```
mailAddress = "myemail@mydomain.com";

pos1 = mailAddress.indexOf("@");

pos2 = mailAddress.indexOf(".");

if (pos1 >= 0 && pos2 >= 0) {

    alert("This is a valid e-mail address!");

}
```

The method *indexOf* returns the first position of the text searched for within the string. In JavaScript, first character in the string has position 0, the second one 1, etc. So if you searched for "m", *indexOf* method would return 0. If the searched text isn't found, the method would return -1.

```
if (mailAddress.indexOf("@") == -1) {

    alert("Character @ wasn't found!");

}
```

You are not limited to searching for characters, as you can even search for whole parts of the string.

```
if (mailAddress.indexOf("mydomain.com") >= 0) {

    alert("You're using mydomain.com!");

}
```

To get the length of a string, just use the *length* attribute.

```
len = mailAddress.length;
```

As you see here, *length* is an attribute of a string, so it doesn't use parentheses (as opposed to *indexOf* or *toLowerCase* which are methods, so they use parentheses).

One more useful method for extracting a part of the string, is the *substr* method. It has two parameters – the first one defines the position of the starting character, and the second one defines the length. For example, if you want to extract the first three characters of a string:

```
firstThree = mailAddress.substr(0, 3);
```

We've used 0 as the first parameter, as the extraction starts from the first character.

How would you extract only the username from the e-mail address? For example, if the address is "myemail@mydomain.com", how would you get only "myemail"? It's easy – first you need to find the position of the character "@" which separates the user name from the domain name and then you need to extract everything that's left from that position.

```
username = mailAddress.substr(0, mailAddress.indexOf("@"));
```

The method *indexOf* returns the number 7 in this case, and since we're starting the count from 0, that's exactly the number of characters we want to extract. If we wanted to include the character "@", we would have to increment the number of characters by one. Check this for yourself – look at the indexes of characters:

```
0 1 2 3 4 5 6 7 8 9 ...

m y e m a i l @ m y ...
```

Document object

The document object is used to represent the current web page that your script is running on. Since in HTML there is no DOCUMENT element, the *document* object in JavaScript contains a few different attributes of the BODY element and it also includes most of the elements contained within the body of the page.

In this guide, we'll try to keep it simple and leave the more advanced stuff for later, so now we'll show you some basic, but useful tricks. In the next example you'll see how to dynamically change the background color of the page. Although there is a more usable way to do it (for example, letting the user choose from a dropdown menu), just for the sake of an example, here's how it's done with users direct input.

```
color = prompt("Enter HTML code for the page background:", "#");

document.bgColor = color;
```

To change the background color of the document, just use its *bgColor* attribute. You can define colors using hexadecimal code or standard names.

```
document.bgColor = "black";

document.bgColor = "#000000";
```

It is important to note that the page will have the background color as defined in its HTML code until you change it in JavaScript code. This change can happen at any time – after a certain event, or you can, for example, let the user decide.

```
if (confirm("Do you want to change the background color?")) {

    document.bgColor = "#ABCDEF";

}
```

We've used *confirm* method that opens a standard Yes / No window with the supplied message. If the user clicks Yes, the *confirm* method returns *true*, otherwise it returns *false*.

Instead of using the *alert* method to display messages to the user, you can write them directly to the page using the *write* method. This way you can add content to the web page through JavaScript. If the user looks at the page source, he won't see this content, as it is added dynamically with JavaScript code.

```
<b>
<script type="text/javascript" language="javascript">
document.write("Bold text.");
</script>
</b>
```

This way you can add text to the page dynamically, based on the user's input, for example:

```
if (confirm("Do you want to add text?")) {
    document.write("Text added!");
}
```

This text is added right to where the SCRIPT element is positioned on the page. So, obviously, if you want to add content using the *document.write* method, you need to place the SCRIPT element within the HTML BODY tag at the appropriate location.

To wrap up, let's just show a couple more interesting attributes of the *document* object. The *lastModified* attribute can, just as its name says, tell you when your page (or again, the document) was last modified. We'll write it directly to the page:

```
document.write("Last modified: " + document.lastModified);
```

It's the same property that is displayed in Internet Explorer if you click *File – Properties* and check when the page was last modified.

The *title* attribute gives you access to the title of the page, the text written inside the TITLE element. You can even change it:

```
alert("Current page title: " + document.title);


document.title = "My new title!";
alert("New page title: " + document.title);
```

After you change the title of the page through code, that change won't be visible in the page source (if the user clicks View – Source in Internet Explorer). The source will still be showing the original title which was hardcoded in the HTML tags. This is because you have actually changed the title dynamically after the page has already loaded.

Quiz 4

Question 4.1: How would you display the following message to the user?

```
He said: "I can't let that happen!"
```

Answer 4.1: Use escape characters to display quotes. There are two ways, depending on which quotes (single or double) you decide to use with the *alert* method.

```
alert("He said: \"I can't let that happen!\".");
alert('He said: "I can\'t let that happen!".');
```

Question 4.2: Write the code to make a simple check if the URL the user has entered is valid (typical URL: "http://www.microsoft.com").

Answer 4.2: We'll check three URL properties. The left part of the URL must be "http://", the URL must also contain a dot (which is at least one character after the "http://" string) and it must be more than 11 characters long (the simplest URL entered can be "http://a.bc").

```
url = prompt("Enter URL:", "");
// Extract few URL properties
leftPart = url.substr(0, 7);
posDot = url.indexOf(".");
urlLen = url.length;
// Check if it is valid
if (leftPart == "http://" && posDot > 7 && urlLen >= 11) {
    alert("URL is valid.");
} else {
    alert("Please enter URL in the following form:\n\n
http://www.domain.com");
}
```

Question 4.3: How would you write text with the following formatting (bold and regular) on the page?

```
Regular text. Bold text. Regular text.
```

Answer 4.3: With the *document.write* method you can also write HTML code, not just regular text. So if you want to write HTML formatted text on the page, just put the HTML code in the *document.write* method. This is just a very simple example:

```
document.write("Regular text. <b>Bold text</b>. Regular text.");
```