HTML.net

JavaScript Tutorial

JavaScript Tutorial - Table of contents

By *Maria Antonietta Perna*

- Introduction
  A brief introduction to the tutorial and what you can expect to learn.
- Lesson 1: What is JavaScript?
  Learn about JavaScript, who created it, what JavaScript can and cannot do.
- Lesson2: Your First JavaScript
  Learn how to include JavaScript in your web page, how to use comments in your code, and write your first JavaScript script.
- Lesson 3: Events
  In the previous lesson you encountered the *onload* event and used it to check whether JavaScript was working in your browser. Here you learn what events are, why they are important to control your web page behavior with JavaScript, and which core events to look out for in your scripts.
- Lesson 4: Variables and Constants
  A JavaScript program is mostly about processing data. Variables and constants are the building blocks of programming. In this lesson you will learn how to create variables and constants to store and manipulate your data.
- Lesson 5: Smarter Scripts with Operators
  Every program, even the simplest one, contains some logic. In this lesson you will learn how to use mathematical, logical, and comparison operators to give some intelligence to your web page.
- Lesson 6: Even Smarter Scripts with *if...else* and *switch*
  We'll be looking at how you can instruct your web page to make choices on the basis of some given condition.
- Lesson 7: Leave Boring Repetitive Stuff to JavaScript with Loops
  Loops can repeat parts of a script. In this lesson, we look at loops such as *while* and *for*.
- Lesson 8: Package your JavaScript Code with Functions
  In previous lessons you applied core programming concepts to make your web page behave as you wanted to. You also used JavaScript built-in functions such as *alert()* and *document.write()*. Now you will learn to create your own functions to better organize your code.
- Lesson 9: A Gentle Introduction to Objects
  Modern programming languages such as JavaScript, PHP, etc., all support a programming model called **Object Oriented Programming (OOP)**. You've already used variables and constants to store your data; objects are just another way of storing data and of giving you more power to manipulate that data. You will be introduced to core JavaScript built-in objects in the following lessons.
- Lesson 10: JavaScript Objects - Strings
  You've been using the **string** (text) object since the beginning of these lessons. Here for the first time you will learn how to exploit all its power as a JavaScript

object.

- Lesson 11: JavaScript Objects - Date Object
  Learn to use the **Date object** to work with dates and times dynamically. You will be able to get today's date, learn how to set a specific date, and how to display the time on your web page.
- Lesson 12: JavaScript Objects - Math Object
  Performing complex mathematical calculations with JavaScript is a breeze. You will be able to use the Math object to build a square root calculator application.
- Lesson 13: JavaScript Objects - Arrays
  In this lesson, you will learn what an array is, how it is used, and what it can do.
- Lesson 14: JavaScript Living Space - the Browser Environment
  Your JavaScript code comes alive in the context of a browser. Learn more about how the JavaScript language conceptualizes its own environment.
- Lesson 15: Useful Tasks (I) – Setting and Retrieving Cookies
  At this point you're ready to explore how JavaScript performs some real world tasks. Here is how you can **set and retrieve cookies**, that is, small text files that store visitors' preferences for personalization purposes.
- Lesson 16: Useful Tasks (II) – Form Validation
  Our second real world task is so widely known that couldn't have been overlooked. Learn how to ensure website visitors fill out your form correctly.
- Lesson 17: JavaScript Timing Events
  In this lesson you will learn how to write JavaScript code that executes after a certain time interval. As a practical application of this technique, think about those nice photo galleries where a photo fades out as the next one fades in at specified time intervals.
- Lesson 18: Making AJAX Calls
  AJAX (Asynchronous JavaScript and XML) is the buzzword of 21st century web applications, and JavaScript is a big player. In this lesson you learn how to make your web page behave smoothly and responsively just like a desktop application.
- Lesson 19: Short Introduction to jQuery
  No JavaScript course could be complete without at least an introduction to the most widely used JavaScript library of our time, **jQuery**. Learn how to include jQuery in your projects and start using it to easily access elements and events in your web page.
- Lesson 20: Cool Animation Effects with jQuery
  Learn how easy it is to add bells and whistles to your web page with jQuery.
- Lesson 21: Easy AJAX Calls with jQuery
  In lesson 18 you learned to make AJAX calls the hard way. In this lesson you are going to perform the same task the jQuery way, that is, the fast and simple way.

Introduction

By *Maria Antonietta Perna*

JavaScript gives you the freedom to add interactivity and responsiveness to your web pages.

The aim of this tutorial is to provide you with a thorough, yet accessible introduction to JavaScript using **snappy explanations** and **practical tasks** to try out right from the start.

No prior knowledge of JavaScript is assumed, but because JavaScript sits within and manipulates web pages, in order to be able to follow along, you should already be familiar with **HTML** and **CSS**. If you are new to either or both, you're advised to step through our HTML and CSS tutorials first.

JavaScript is a lightweight, easy to learn, scripting language. It's used on almost every website to respond to user actions, validate web forms, detect browser support, and much more.

JavaScript is a web programming language, that is, a language that enables you, the designer of your website, to control how a web page **behaves**. This makes JavaScript crucially different from HTML, the language that gives **structure** to your web documents, and CSS, the language that controls the **appearance** of web pages.

If you know other programming languages such as PHP, most programming concepts and basic JavaScript syntax will sound quite familiar to you. However, if this is not the case, don't worry: by **following along** and **experimenting with the code**, at the end of this hands-on tutorial you'll be able to spruce up your static web pages with fun effects and fantastic responsiveness for the joy of your website visitors.

If you need help along the way, don't forget to turn to our forums. This is where you meet the real experts who are willing and ready to offer tips, suggestions and advice.

What is needed?

To bring to life your web pages with JavaScript all you need is a text editor and an updated, standard-compliant browser of your choice, such as Internet Explorer 9, Firefox, Chrome, Safari, or Opera, to mention just the most popular ones.

Basic free text editors that ship with your operating system such as Notepad (on Windows) and Text Edit (on Mac) will do just fine. However, text editors that offer programming languages support, such as the free Notepad++ (for Windows users) and TextWrangler (for Mac users), might be a great resource when ploughing through more than a few lines of code.

Ready to supercharge your web pages with JavaScript? Let’s get started!

Lesson 1: What is JavaScript?

By *Maria Antonietta Perna*

The first thing that creates some confusion about JavaScript is its name. In fact, one of the most common questions students raise when approaching JavaScript is:

> *"Is JavaScript the same as Java?".*

Clearing up this basic but important question is our first order of business in this lesson, but by no means the only one. By the end of this lesson you will also know:

- the difference between JavaScript and Java;
- what JavaScript can and cannot do.

Are JavaScript and Java the same thing?

No, they are not.

Java (developed by Sun Microsystems) is a powerful and much more complex programming language in the same category as C and C++.

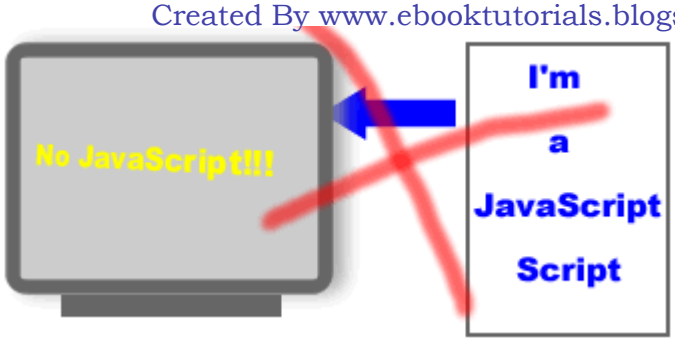
JavaScript was created by **Brendan Eich** at **Netscape** and was first introduced in December 1995 under the name of *LiveScript*. However, it was rather quickly renamed JavaScript, although JavaScript's official name is **ECMAScript**, which is developed and maintained by the**ECMA (European Computer Manufacturer's Association)** International organization.

JavaScript is a scripting language, that is, a lightweight programming language that is **interpreted** by the browser engine when the web page is loaded.

The fact that the JavaScript interpreter is the browser engine itself accounts for some inconsistencies in the way your JavaScript-powered page might behave in different browsers. But don't worry: thankfully, well-established techniques and powerful JavaScript libraries such as **jQuery** (which will be introduced in later lessons) are here to make things wonderfully easier on us.

Things you *can't* do with JavaScript

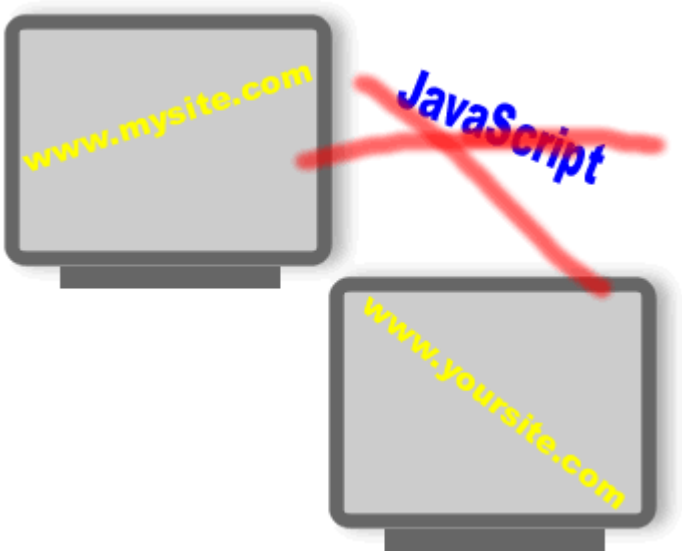
You can't force JavaScript on a browser.



JavaScript runs in the **client**, that is, the brower. If you use an older browser without support for JavaScript, or if you simply choose to disable JavaScript in your browser, then a JavaScript script can't work.

Conclusion: unlike what happens with languages that run on the **server**, such as PHP, you never fully know for sure the impact that the browser your website visitors are going to use will have on your script, or whether your visitors will choose to turn JavaScript support off.

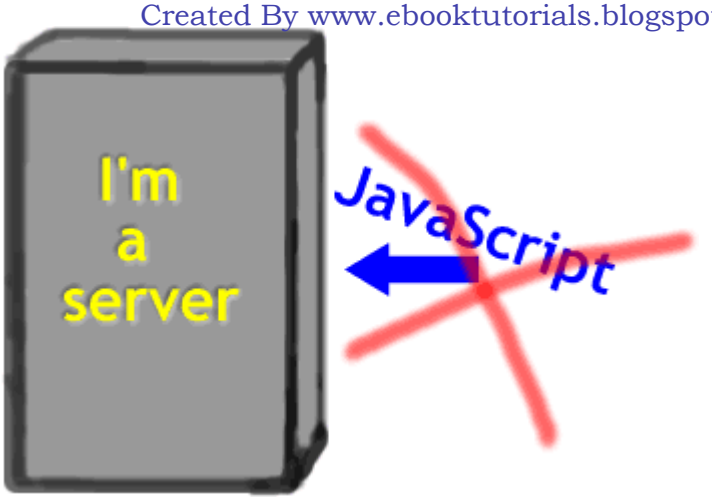
You can't access or affect resources from another internet domain with JavaScript.



This is called the **Same Origin Policy**. Well, how would you like it if all of a sudden all the nice comments your visitors left on your website started to disappear, or to change place in your page because of a naughty JavaScript script running on another website?

This is exactly the kind of nasty situation that the Same Origin Policy is designed to prevent. Conclusion: your JavaScript script can only access resources in your website.

You can't access server resources with JavaScript.



Because JavaScript is a client-side language, it's limited to what can be done in the client, that is, usually in the browser environment. A JavaScript script cannot access server resources such as databases.

Zillion things you *can* do with JavaScript

With JavaScript you can:

✓ Put text in an HTML page on-the-fly.

Say you want to display a nice thank you message to a user who has just submitted a comment form on your website. Obviously, the message needs to be added **after** the user has submitted the form.

You could let the server do that. However, if your website is very busy and your server processes hundreds of forms a day, it might take a little while for your thank you message to appear to the user.

Here's JavaScript to the rescue. Because JavaScript runs in the user's browser, the thank you note can be added and displayed on the page almost instantaneously, making your website users happy.

✓ Make your web pages responsive.

Web environments are dynamic, things happen all the time: the web page loads in the browser, the user clicks a button or moves the mouse over a link, etc. These are called **events** (which will be the topic of lesson 3).

With JavaScript you can make the page immediately react to these events the way you choose: for example, by showing or hiding specific page elements, by changing the background color, etc.

✓ Detect visitors' browsers.

You can use a JavaScript script to detect the visitor's browser, or, even better, you can detect what features a certain browser does or does not support. Depending on the browser and its capabilities, you can choose to load a page specifically tailored to that kind of browser (lesson 14).

 **Create cookies.**

A JavaScript script is great if you want to create cookies so that your visitors can enjoy a personalized experience the next time they visit your website (lesson 15).

 **Validate web form data.**

You can use a JavaScript script to validate form data before the form is submitted to a server. This saves the server from extra processing (lesson 16).

And much ... much more.

Learning JavaScript will enable you to add cool animation effects to your web pages without using an external Flash plug-in, use the newest features of HTML5 such as canvas (to draw directly on your web page) and drag and drop capabilities, integrate your website with external web services such as Facebook, Twitter, etc.

Summary

In this lesson you learned **what JavaScript is**, **who invented it**, and that the body responsible for its maintenance and continuous development is **ECMA**. Now you know **what you cannot do with JavaScript**, but also the **great things that you can do with it**.

At this point you might say:

> "That's all well and good. I'm convinced, JavaScript is fantastic. But, how can I start using it in my HTML page?"

That's easy: learn how this is done in the next lesson.

Lesson 2: Your First JavaScript

By *Maria Antonietta Perna*

Now that you know what JavaScript is and what you can do with it, it's time to get to the practical stuff.

In this lesson you are going to learn:

- how to embed your JavaScript in the HTML page;
- how to reference your JavaScript from a separate file;
- how to comment your JavaScript code and why it is recommended;
- how to create your first JavaScript-powered web page.

The HTML

To insert a JavaScript script in an HTML page, you use the **<script> ... </script>** tag. **Don't forget the closing </script> tag!** Now get ready to fire off your text editor of choice and let's get coding!

Let's start with a basic HTML page, like this one:

```
<!DOCTYPE html>
<html>
<head>
<title>My first JavaScript page</title>

</head>
<body>

</body>
</html>
```

The JavaScript script is inserted either in the HTML page itself or in a separate file.

Embed JavaScript in the HTML page

The <script> tag and its *type* attribute tell the browser: "Hey, browser! There's a script coming up, and it's a JavaScript script."

You can do this either in the <head> section, as follows:

```
<!DOCTYPE html>
<html>
<head>
<title>My first JavaScript page</title>

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

```
</script>

    </head>
    <body>

    </body>
    </html>
```

Or or at the very bottom of the document just before the closing </body> tag, like so:

```
    <!DOCTYPE html>
    <html>
    <head>
    <title>My first JavaScript page</title>
    </head>
    <body>

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

    </body>
    </html>
```

If you're wondering whether it's best to place your <script> tag in the <head> or the <body> tag, then you're not alone.

It mostly comes down to personal preference. However, because most of the times you'll want your JavaScript code to run **after** the web page and all its resources, e.g., stylesheets, graphics, videos, etc., have finished loading in the browser, I suggest you dump your JavaScript <script> tag at the bottom of your page.

Comments, comments, comments

One final thing to note about both code snippets above is the two forward slashes ***//*** before the text "JavaScript code goes here". This is how you **comment a one-line JavaScript code**.

When a comment spans over more than one line, you use ***/* Comment goes here */*** to delimit a comment, just like you do in a stylesheet. Here's how it's done:

```
    <!DOCTYPE html>
    <html>
    <head>
    <title>My first JavaScript page</title>
    </head>
    <body>

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

    </body>
    </html>
```

When the JavaScript interpreter in your browser comes across either '//' or '/* */', it just ignores whatever text is placed in between. Use comments in your code to remind your future self of what your code is designed to do. One day you'll be happy to have done so, just take my word for it!

Insert JavaScript in a separate file

If your script is longer than a few lines, or you need to apply the same code to several pages in your website, then packaging your JavaScript code into a separate file is your best bet.

In fact, just like having your CSS all in one place, well away from HTML code, having your JavaScript in its own file will give you the advantage of easily maintaining and reusing your scripts.

Here's how it's done:

```
    <!DOCTYPE html>
    <html>
    <head>
    <title>My first JavaScript page</title>
<script type="text/javascript" src="yourjavascript.js"></script>
    </head>
    <body>

    </body>
    </html>
```

As you can see, the *<script>* tag references an external file, **"yourjavascript.js"** that has the **.js** extension and contains the script that puts the magic into your web page.

Your first JavaScript-powered page: Hello World

Without further ado, let's see if JavaScript works in your browser.

Try out: embedded JavaScript in an HTML page

Between the <script> and </script> tags either in the <head> or the <body> of your HTML document, add the following line of code, just after the comment:

```
<!DOCTYPE html>
<html>
<head>
<title>My first JavaScript page</title>
</head>
<body>

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

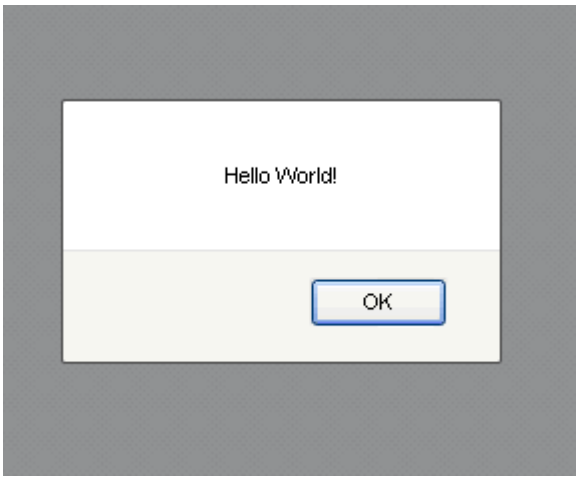
```
    alert('Hello World!');
</script>

    </body>
    </html>
```

This is your first JavaScript **statement**, that is, you've just instructed your web page to do something. Don't worry about the code just yet, we'll be coming back to the **alert()** command again and again in the following lessons.

Just notice the *semicolon (;)* at the end of the statement. This is important: it tells the JavaScript interpreter that the statement is finished and whatever comes next is a different statement.

Now save your work and run the page in your favorite browser. You'll see an alert box popping up on page load. This is how the page looks in Firefox:



If your alert box is not popping up, then check that you've typed the JavaScript command exactly as it is in the sample code.

Make sure your **<script>...</script>** tags are there, that the text between the brackets is **surrounded by quotes (' ')**, and that **there is a semicolon (;) at the end of the statement**. Then try again.

Try out: JavaScript in a separate file

Create a new document in your text editor and save it as "helloworld.js". **Important: the file extension has to be .js** (this is the appropriate JavaScript file extension).

In the new document, paste in the JavaScript command from the previous example (**no need** to type the <script> ... </script> tags here):

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

Now, go back to the HTML page, delete the previous JavaScript code, and add the

<script> ... </script> tags in the <head> section of the page with a reference to the helloworld.js JavaScript file, like so:

```
<!DOCTYPE html>
<html>
<head>
<title>My first JavaScript page</title>
<script type="text/javascript" src="helloworld.js"></script>
</head>
<body>

</body>
</html>
```

Save all your documents and run the HTML page in the browser. You should view the same alert box popping up as in the previous example.

If the code doesn't work, in your HTML document check that the filepath to the JavaScript file is correct, the filename spelling is accurate, and double-check that you've added a closing </script> tag. In helloworld.js make sure your JavaScript command is typed exactly the same as in the sample code above, then try again.

Summary

You've actually learned a lot in this lesson. You know **how and where to include JavaScript** in your web page, how to **comment your JavaScript code and why this is a good idea**, and finally you've just seen your web page come alive with **your first JavaScript script**.

Admittedly, an alert box saying "Hello World!" looks a bit dumb, but even alerts can be useful to quickly and easily test that JavaScript is enabled in your browser and your code is working.

It's time for a well deserved break. In **lesson 3** you'll be tackling another core topic in your JavaScript journey: **events**. Get ready!

Lesson 3: Events

By *Maria Antonietta Perna*

The web is a dynamic environment where a lot of things *happen*. Most appropriately, they're called **events**. Some of these events, like a user **clicking** a button or **moving the mouse** over a link, are of great interest to a JavaScript coder.

By the end of this lesson you will know:

- what events are;
- **what** JavaScript can do with events and **how**.

What are events?

Events are occurrences taking place in the context of the **interactions** between **web server**, **web browser**, and **web user**.

For instance, if you're on a web page and click on a link, there are at least 3 important events being triggered:

1. **onClick** event: triggered by you as you click the link;
2. **onUnload** event: triggered by the web browser as it leaves the current web page;
3. **onLoad** event: triggered by the browser as the new web page content is loaded.

Which events should I focus on from a JavaScript point of view?

It all depends on the JavaScript program you're writing, that is, on the objectives you want to achieve with your script.

However, the most common events you're likely to deal with in your JavaScript are:

- *onLoad/onUnload*;
- *onClick*;
- *onSubmit* (triggered by the user submitting a form);
- *onFocus / onBlur* (triggered by a user as, for example, she clicks in a textbox or clicks away from a textbox respectively);
- *onMouseOver / onMouseOut* (triggered by the user moving the mouse over or away from an HTML element respectively).

There are other events that might eventually be of interest to you as you write sophisticated JavaScript scripts, such as **scroll** events (as users scroll up and down a web page), **onTextChanged** events (as users type in a textbox or textarea), even **touch** events, which are likely to gain interest in today's mobile and tablet-invaded world.

However, for the purposes of this tutorial, you're mostly going to come across the core events listed above.

What do I do with events from a JavaScript point of view?

As you write a JavaScript program, events become interesting because they give your script a hook for gaining control on what happens in the web page.

Once your script gets hold of the hook provided by the event, your script is boss. The jargon for this is **event-driven programming**: an event happens and JavaScript **handles** it, for instance by **displaying an alert box with a message for the user**.

Hey, an onClick event going on!

Conclusion: events bend to JavaScript commands by means of **event handlers**. These are statements in your JavaScript script that are appropriately **attached** to those events.

How does JavaScript handle events?

In this tutorial you've already hooked an event handler to an event. More precisely, you

attached the **alert()** command to the **onLoad** event.

The **alert()** command is a command which is part of the JavaScript language. The JavaScript interpreter in the browser translates it along these lines:

> "Hey, browser, display an alert box that contains the message typed within the () enclosing brackets"

(**Notice: JavaScript is case sensitive**. If you write **A**lert instead of **a**lert, your script won't work!).

As you saw in the previous lesson, simply by writing a JavaScript statement between <script> ... </script> tags achieves the **execution of that statement** as the **onLoad** event fires up.

However, your JavaScript scripts are capable of doing more interesting stuff when they handle events in response to user actions, for example an onClick event. How do we do that?

Try out: handle an onClick event with JavaScript

The browser already has its own ways of **handling events**. For instance, when a page has loaded, the browser fires the onLoad event and displays the page contents; when a user clicks a link, the browser communicates to the server to access the requested page, etc. These are called **default actions**.

The fun of being in charge, though, is **not to let the browser do what it likes**, but of **letting JavaScript do its job and decide what's to be done**.

The simplest way to attach an event handler to an event is to insert the required JavaScript code within the HTML element that produces the event. Let's have a go by simply preventing the browser default action as the user clicks a link on the page. Fire off your text editor and let's get coding!

Prepare a new basic HTML document displaying a simple link like the one shown below:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 3: Events and Event Handlers</title>
</head>
<body>
<h1>Lesson 3: Events and Event Handlers</h1>

<a href="http://html.net">Click Me!</a>

</body>
</html>
```

Run the page in the browser. If you click on the link now, you'll be landing straight to the HTML.net website. This is the browser default action, as explained above.

Now go back to your HTML document and type the following JavaScript command within the **<a>** tag, as follows:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 3: Events and Event Handlers</title>
</head>
<body>
<h1>Lesson 3: Events and Event Handlers</h1>

<a href="http://html.net" onclick="alert('Going anywhere? Not so
fast!'); return false;">Click Me!</a>

</body>
</html>
```

Go ahead, run the page in the browser, and ... **you're stuck! JavaScript is in control!**

What's just happened there?

That's how easy it was to take control of a link with JavaScript! All you did was to insert a couple of **JavaScript statements** to the **onclick** attribute of the HTML <a> tag.

You already know about the good old alert() command, so I'll skip over it. The **return false;** command tells the JavaScript interpreter to return a value, in this case the value equals false, which prevents the browser from performing its default action. You'll be using this little command quite often in your JavaScript programming life.

You handle the other events listed above in the same way: just insert the JavaScript command as the value of the onfocus, onblur, onmouseover, onmouseout and onsubmit attributes of an HTML element.

Summary

That'll be all for this lesson. You learned what events are, why events are important in JavaScript programming, and how to use event handlers to let JavaScript be in control of your web page behavior.

But this is just a tiny taste of what JavaScript can do once it's in charge. Follow on to find out.

Get ready for the major topic of lesson 4: **variables** and **constants**.

Lesson 4: Variables and Constants

By *Maria Antonietta Perna*

Variables and **constants** are like containers where you store data and values for processing in JavaScript.

The difference between a **variable** and a **constant** is this: once you give a **value** to a **constant the value is meant to be kept unchanged** throughout the script. In all circumstances where you reasonably foresee that the **original value is modified** through the script, by all means use a **variable** as your storage room. In fact, **you're going to use variables most of the times**.

In this lesson, you're going to learn:

- how to **create** variables and constants;
- how to **assign** values to variables and constants;
- how to **use** variables in your code;
- how to **name** variables and constants correctly.

How to create variables and constants

You **declare**, that is, create variables and constants in a similar way.

Here's how you **declare a variable**:

```
/*To declare a variable you give it a name preceded
by the JavaScript keyword var*/
var amountDue;
```

In the sample code above you declare a variable named *amountDue*.

The **variable declaration** ends with a (;) semicolon, which makes it a self-contained statement.

Also, because JavaScript is **case sensitive**, every time you refer to the *amountDue variable*, make sure you **keep the letter case exactly the same as the original declaration** (this particular notation is called **camelCase** because it looks like a camel's hunch).

Here's how you **declare a constant**:

```
/*To declare a constant you give it a name preceded
```

```
by the JavaScript keyword const

Also take note: const is not supported by Internet Explorer

use var instead: it's safer*/

const taxRate;
```

In the sample code above, you declare a taxRate constant. As you can see, the declaration is very similar to the variable declaration, the only difference being the keyword used: **var in the case of a variable** and **const in the case of a constant**.

How to assign values to variables

After you create a variable, you must **assign** (give) a value to it (or said in Geeeky talk **"initialize a variable")** . The **(=) equal sign** is called **assignment operator**: you assign the value of what is on the right side of the = sign to whatever is on the left side of the = sign. **Notice: you cannot perform operations with empty variables.**

Ready to fire off your text editor? Let's get coding!

Prepare a basic HTML document with the JavaScript code illustrated below:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 4: Variables and Constants</title>
</head>
<body>
<h1>Lesson 4: Variables and Constants</h1>

<script type="text/javascript">

//Create a variable

var amountDue;

/* Assign a value to it:

you do not know the value yet

so for now it is 0 */

amountDue = 0;

/* Create 2 more vars and assign

a value to each at the same time */

var productPrice = 5;

var quantity = 2;

/* Assign a value to amountDue by

multiplying productPrice by quantity */

amountDue = productPrice * quantity;

/* Notice how the value of amountDue has
```

```
changed from 0 to 10 -

-Alert the result to check it is OK

Notice: you do not use ' ' with var name in alert() */

alert(amountDue);

</script>

</body>
</html>
```

Did you see the good old alert box displaying the number 10 popping up? If you didn't, make sure your code is typed exactly the same as in the snippet above (**Notice: When tracing bugs (errors) in your JavaScript code, look out for brackets, semicolons (;), quotes (' '), and letter casing**).

That's great! Try experimenting with the code above. For example, change the value of quantity and productPrice, or just come up with your own variables.

How to name variables (and constants)

Choose **descriptive names** for your variables (**var amountDue;** rather than **var x;**): this way your code will be more **readable and understandable**.

While this can be called good programming practice (but also common sense), there are also **syntax rules** (yes, just like any natural language) when it comes to naming variables, and they must be obeyed, at least if you want your JavaScript script to work.

Keep an eye on the following simple rules when you name variables:

1) The first character must be a letter, an (_) underscore, or a ($) dollar sign:

5total is wrong: var name cannot start with a number

2) Each character after the first character can be a letter, an (_) underscore, a ($) dollar sign, or a number:

3) Spaces and special characters other than (_) and $ are not allowed anywhere:

- to tal is wrong: spaces are not allowed;
- total# is wrong: # character is not allowed;
- total£ is wrong: £ character is not allowed.

Summary

Variables and constants are the building blocks of any programming language. In this lesson you learned the part they play in JavaScript, how you declare and assign values to them, how to name your variables correctly, and you also had a taste of using variables in JavaScript.

In the next lesson you're going to learn about **JavaScript operators** and do more and more practice with variables.

Take a break and get ready for lesson 5.

Lesson 5: Smarter Scripts with Operators

By *Maria Antonietta Perna*

In the previous lesson you already employed an **assignment operator (=)** and an **arithmetic operator**, specifically the **multiplication operator (*)**, to write a basic JavaScript shopping cart script.

We can easily see that to do something useful with JavaScript, we need a way to **manipulate** data and variables. We do this with **operators**.

In this lesson you are going to learn how to use:

- **arithmetic operators**;
- **the + sign to concatenate text (concatenation operator)**;
- **comparison operators**;
- **logical operators**.

Also, you'll get plenty of opportunities to practice coding with variables. Let's get started!

Arithmetic operators

As you might have guessed, **arithmetic operators** are used to **perform arithmetic operations between values or variables**. Here's a table for your reference.

If **x = 20, y = 5, and z = result**, we have:

Operator	Java Script Example	Result
Addition: +	z = x + y	z = 25
Subtraction: -	z = x - y	z = 15
Multiplication: *	z = x * y	z = 100
Division: /	z = x / y	z = 4
Modulus: %	z = x / y	z = 0
Increment: ++	z = ++x	z = 21
Decrement --	z = --x	z = 19

I guess you're quite familiar with most arithmetical operators. The odd ones might be the **(%) modulus**, the **(++) increment**, and the **(--) decrement** operators.

Modulus: the **remainder** left over after division.

Increment: take a number and add 1 to it.

Decrement: take a number and subtract 1 from it.

Time to get coding! Get your hands on the text editor and prepare a new HTML document like the one below:

Try out: add 2 values and print the result

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 5: Operators and Comparisons</title>
</head>
<body>
<h1>Lesson 5: Operators and Comparisons</h1>

<script type="text/javascript">

//Create and initialize your variables

var result = 0;

var firstNum = 20;

var secondNum = 5;

//Addition: result = 25

result = firstNum + secondNum;

//write result on the page

document.write(result);

</script>

</body>
</html>
```

Nothing new here except for the JavaScript command **document.write()**. This command is translated by the JavaScript interpreter as saying:

> "Hey browser, get the value within brackets and print it on the HTML document!"

In our case the value is a variable, therefore no (' ') quotes are used to enclose it. If you want to print some text instead, the command must be: **document.write('some text.');**. It's all very similar to the alert() command you've been using so far.

Now **experiment with the code** sample above by trying out all the other arithmetic operators and printing the result on the page.

Concatenation operator

If you want to add pieces of text together to form one long line of text, use the + sign. In Geeky talk a piece of text is called **string**, and it appears enclosed either in (' ')

quotes or (" ") double-quotes (remember the 'Hello World' text you used in the alert() command? That is an instance of string).

Try out: concatenate strings and print a message on the page

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 5: Operators and Comparisons</title>
</head>
<body>
<h1>Lesson 5: Operators and Comparisons</h1>

<script type="text/javascript">

//Create and initialize your variables

var firstText = "Hello";

var secondText = "World!";

//Resulting value of assignment is Hello World!

var message = firstText + ' ' + secondText;

//write result on the page

document.write(message);

</script>

</body>
</html>
```

If you typed your code correctly, you should see the famous *Hello World!* text smack on the web page. **Notice: you separate** *Hello* **and** *World!* **by concatenating quotes (' ') in-between each piece of text or variable.**

Now get some **practice concatenating strings** before moving on.

Comparison operators

Often you need to compare different values and make your JavaScript program take different directions accordingly.

For example, you're coding a JavaScript script for a shopping cart application. At one point, your script will have a statement saying something along these lines: if the total amount to be paid is **greater than or equal to** $50 apply a 5% discount, if it's **less than or equal to** $50 do not apply 5% discount. Don't be impatient, **you will learn how to code this kind of conditions in the next lesson**.

It's here that **comparison operators**, such as **equal to, less than**, etc. enter the scene. Here below are listed all comparison operators for your reference.

If x = 10 we have:

Operator	What is it?	Example
==	equal to	x == 5 is false
===	exactly equal to value and type	x === 10 is true x === "10" is false
!=	not equal	x != 2 is true
>	greater than	x > 20 is false
<	less than	x < 20 is true
>=	greater than or equal to	x >= 20 is false
<=	less than or equal to	x <= 20 is true

Logical operators

You use **logical operators** when you need to determine the logic between certain values.

Going back to the shopping cart script example, you might want your script to apply a 5% discount **if the following 2 conditions are both true**: a given product costs **more than** $20 **and** is purchased before the 31st of December.

Here come **logical operators** to the rescue. Given that x = 10 and y = 5:

Operator	What is it?	Example
&&	and	(x < 20 && y > 1) is true both conditions must be satisfied
||	or	(x == 5 || y == 5) is true at least 1 condition must be satisfied
!	not	!(x == y) is true

Questions, questions, questions

The tables above are self-explanatory, except for the following 2 questions:

1. When you talk about === , what do you mean by equality of **value** and **type**?
2. What's the difference between **(=)**, **(==)**, and **(===)** ?

Answer to question 1.

Values are the specific data, either directly in your JavaScript statements or contained in JavaScript variables. For example:

```
var price = 5;
```

In the code snippet above, the variable *price* has value *5*.

What's the **type**?

The type, or more precisely the **data type**, is the way JavaScript classifies data. You've come across 2 data types, that is, **number and string (text)**. A third data type is **Boolean**, that is, **true and false statements**.

Therefore, when you compare 2 values using (===), the 2 values are compared on the basis of **both their value and their data type**:

```
var firstNum = 4;
var secondNum = 4;
//this is true: both values are 4
//and both values are of type number
firstNum === secondNum;
//let's use a string data type. A string uses ' '.
var stringNum = '4';
//Now === is false: 4 and '4' are different types
firstNum === stringNum;
```

Answer to question 2.

The (=) operator is used to **assign or give** a value to a variable. It is **not a sign for equality**.

The (==) and (===) operators instead, **do stand for equality**. They **do not assign values to variables**. **(==) compares only values, (===) compares both values and data type**.

Summary

You've made it all the way through this lesson, congratulations! Now your scripts are not limited to just popping up messages. They start to **be smart**: they can **make calculations, comparisons, and set truth conditions to evaluate their data**.

In the next lesson you keep adding intelligence to your JavaScript with *if ... else* and *switch* statements.

Take a break and get ready for it!

Lesson 6: Even Smarter Scripts with if... else and switch

By Maria Antonietta Perna

Now that our JavaScript code can **calculate, compare, and determine true and false conditions**, things start looking a lot more interesting.

The days of dumb message boxes popping up are long gone, the new era of smart message boxes has begun.

In this lesson you're going to inject the power of choice into your scripts with **conditional statements**. These statements enable your script to evaluate given conditions on the basis of which to execute one or more actions. More specifically, you're going to learn how to use:

- **if** statements;
- **if ... else** statements;
- **if ... else if ... else** statements;
- **switch** statements.

In the process, you're also going to learn how to use **shortcut assignment operators**.

if Statement

The **if** keyword is used in JavaScript to perform a basic test for the **truth or falsity of a given condition and execute a piece of code accordingly**. Here's its structure:

```
//if the condition in brackets is true
if (condition)
{
//do some stuff here
}
```

Let's translate the code

Translated into English, the **if statement** above says:

> "Hey browser, if the condition in the round brackets is true, execute the commands between those odd curly braces. If the condition is not true, then ignore everything and move on!"

Important: do not forget either the brackets ( ) or the braces { }.

if ... else statement

Use this statement to **execute a command if the condition is true** and **another command if the condition is false**. Here's its structure:

```
//if the condition is true
if (condition)
{
//do this stuff
}
//otherwise
else
{
//do this other stuff
}
```

if ... else if ... else statement

Use this statement if you want your code to **evaluate several conditions on the basis of which to execute the appropriate commands**. Here's its structure:

```
//if the condition is true
if (condition)
{
// do this stuff
}
//if this other condition is true instead
else if (condition)
{
// do this other stuff
}
//if none of the above condition is true
else
{
// do this instead
```

```
}
```

switch statements

In alternative to an *if ... else if ... else* statement you can switch to a ... well a *switch statement*. Here's its basic structure:

```
//condition to evaluate
switch(condition)
{
//in this case do this
case 1:
execute code block 1
//stop here: no need to keep going
break;
//in this other case do that
case 2:
execute code block 2
//stop here: no need to keep going
break;
//otherwise fall back on this
default:
code to execute when all previous conditions evaluate to false
}
```

It's time to put your new knowledge to work right away. In this try out you're going to check a product price and decide whether to apply a discount. Prepare a new HTML document and get coding!

Try out: check product price with *if*

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 6: Even Smarter Scripts with if…else and switch</title>
</head>
<body>
<h1>Lesson 6: if statements</h1>

<script type="text/javascript">

//store 5% discount value in a var
```

```
var discountRate = 0.05;
//store price of 1 apple in a var with a value of 2
var applePrice = 2;
//store quantity value in a var
var quantity = 5;
//declare a var for the discounted total value
//because we don't know the value we initialize it with 0
var discountedTotal = 0;
//store the value of the total in a var
//you use the ( * ) multiplication operator to calculate the total
//and the ( = ) assignment operator to assign the value to the var
var total = quantity * applePrice;
//use if to check the total is entitled to a discount
//if total is greater or = to 10 it's entitled
if (total >= 10)
{
alert("You're entitled to a discount!");
//apply 5% discount on the total
//and assign it to discountedTotal var
//Take note: multiplication is in brackets because it has to be
//performed before subtraction (remember the good old school math?)
discountedTotal = total - (discountRate * total);
//create a var to store text to display to user
var infoText = "You'll get your delicious apples at $" +
discountedTotal;
//you add more text to the same infoText var with a shortcut assignment
operator
//this technique is useful when you have long text strings to manipulate
infoText += " instead of at $" + total;
//display message to the user
document.write(infoText);
}
</script>
</body>
</html>
```

Save your work, run the page in the browser, and enjoy your discount!

Questions, questions, questions

It's almost like I can hear you saying something like:

> Hey, what's going on there? What's that **+= sign** all about?

Here's the answer:

Shortcut operators

When you add several values to a variable, a shortcut operator is very handy.

The expression **x += 2** is the same as **x = x + 2**.

As you can see, the given value of x is taken as the starting point, 2 is added to it, and the resulting value is reassigned to x. Using a shortcut operator spares you from having to retype x.

When applied to the code snippet above, the expression **infoText += " instead of at $"** is the same as **infoText = infoText + " instead of at $"**. If we simply use an (=) assignment operator in this case, the text already stored in infoText will be wiped out by the new value. But this is not what we want in this script.

Handy shortcuts are also available to all other mathematical operators:

- -=
- *=
- /=
- %=

Try out: check product price with *if … else*

Just after the closing curly brace **}** from the previous example, add the following snippet.

```
else
{
alert("Sorry, you're not entitled to a discount");
var infoText = "Your delicious apples cost $" + total;
document.write(infoText);
}
```

Change the value in the *quantity variable to be less than 5*, save your work and run the

page in the browser. Now, JavaScript should give you the not so good news that you're not entitled to a discount.

Try out: check product price with *if … if else … else*

Still using the previous example, **between the first if block** and the **last else block**, insert the following **else if block**.

```
//if the total equals to 8
else if (total == 8)
{
alert("Just add a couple more apples and get your discount!");
var infoText = "Be a sport, buy a bit more and pay a bit less";
document.write(infoText);
}
```

Change the *quantity variable to be 4*, save your work and run the page in the browser. Now you should fall for the persuasive power of such a great discount.

Your code can detect if the purchase **qualifies for a discount**, if it **almost qualifies for a discount**, and finally if the **discount is inapplicable by a long shot**. In each case, your code can take appropriate action.

Have fun practicing with **if ... else if ... else** statements, and **shortcut operators**.

Try out: days of the week with *switch*

Now get a fresh HTML document ready and type the following code block:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 6: Even Smarter Scripts with if…else and switch</title>
</head>
<body>
<h1>Lesson 6: switch</h1>
<script type="text/javascript">
//pick a day of the week and store it in a var
var today = "Friday";
//check today var value and act accordingly
switch(today)
{
//in case value corresponds to Saturday or to Sunday do this:
```

```
case "Saturday":
      case "Sunday":
alert("It's the weekend: great!");
document.write("Today is " + today);
//no need to keep going:
break;
//in case value corresponds to Friday do this
case "Friday":
alert("Almost weekend time!");
document.write("Today is " + today);
//no need to keep going
break;
//in all other cases do this instead
default:
alert("Just an ordinary weekday: keep up the good work");
document.write("Today is " + today);
}
</script>
</body>
</html>
```

Save your work and run the page in the browser: a great Friday message should be there to greet you. Keep changing weekday and make sure the alert box displays the appropriate message each time.

Summary

Now you know how to write scripts that implement different actions on the basis of different conditions. Well done!

Keep experimenting with all the tools you've learned so far. If you have doubts, drop a line in the forum dedicated to this tutorial. When you're ready, move on to the next big topic: **loops**.

Lesson 7: Leave Boring Repetitive Stuff to JavaScript with *loops*

By Maria Antonietta Perna

Your JavaScript scripts can do all sort of clever stuff like calculating discounts, comparing values, and making choices. However, don't forget we're dealing with machine language, and machines are very good at doing some things us humans would keep a mile off if only we could.

One really boring thing most of us would happily avoid doing is repetitive stuff. You might find it hard to believe this, but JavaScript loves repetition. So much so that it has a special construct for it: the **loop**.

Here's what you will learn in this lesson:

- the **for loop**;
- the **while loop**;
- the **do ... while loop**.

In the process, you will also learn how to:

- use **getElementById(element Id)** to manipulate an HTML element with JavaScript;
- generate **random numbers with random() and floor(number)**;
- insert **HTML mark-up dynamically with innerHTML**.

This is going to be a meaty lesson. Get ready for it!

Loops

At times you will have to repeat some piece of code a number of times:

for instance: a web page has 10 radio buttons and your script needs to find out which radio button was checked by the user.

To do so, your script has to go over each single radio button and verify whether its **checked attribute** is set to true or false. Does this mean having to write this kind of *verification code* 10 times?

Thankfully, it does not. You **dump the code in a loop once, and it's going to execute any number of times** you set it to.

Any loop is made of **4 basic parts**:

1. **the start value**

   An initial value is assigned to a variable, usually called **i** (but you can call it anything you like). This variable acts as counter for the loop.

2. **the end value or test condition**

   The loop needs a limit to be set: either a definite number (loop 5 times) or a truth condition (loop until this condition evaluates to true). Failing this, you run the risk of triggering an **infinite loop**. This is very bad: it's a never-ending repetition of the same code that stops users' browsers from responding. Avoid infinite loops at all costs by making sure you set a boundary condition to your loops;

3. **the action or code to be executed**

   You **type a block of code once and it'll be executed the number of times between your start value and end value**;

4. **the increment**

   This is the part that moves the loop forward: the counter you initialize has to move up (or down in case you opt for looping backwards). As long as the loop **does not reach the end value or the test condition is not satisfied**, the counter is incremented (or decremented). This is usually done using mathematical operators.

The for Loop

This kind of loop, well ... loops through a block of code a set number of times. Choose a **for loop** if you know in advance how many times your script should run.

Here's its basic structure:

```
    //loop for the number of times between start value
    //and end value.  Increment the value each time
    //the limit has not been reached.
    for (var=startvalue; var<=endvalue; var=var+increment)
{
//code to be executed
}
```

The while Loop

If you don't know the exact number of times your code is supposed to execute, use a **while loop**.

With a while loop your code **executes while a given condition is true**; as soon as this condition evaluates to false, the while loop stops.

Here's its basic structure:

```
    //loop while initial value is less than or equal to end value
    //Take note: if the condition evaluates to false from the start,
    //the code will never be executed
    while (variable <= endvalue)
{
//code to be executed
}
```

do ... while Loop

This kind of loop is similar to the *while loop*. The difference between the two is this:

In the case of the *while loop*, **if the test condition is false from the start**, the **code in the loop will never be executed**.

In the case of the *do ... while loop*, the test condition is evaluated **after** the loop has performed the first cycle. Therefore, even **if the test condition is false, the code in the loop will execute once**.

Here's the basic structure of a do ... while loop:

```
    //the command is given before the test condition is checked
    do
    {
    //code to be executed
    }
    //condition is evaluated at this point:
    //if it's false the loop stops here
    while (variable <= endvalue)
```

Try out: add random number of images with a *for*

loop

Time to get coding: prepare a fresh HTML document with a div tag and an id of "wrapper" and add the following JavaScript magic:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 7: Leave Boring Repetitive Stuff to JavaScript with Loops</title>
</head>
<body>
<h1>Lesson 7: for Loop</h1>

<div id="wrapper"></div>

<script type="text/javascript">

//Grab the html div by its id attribute

//and store it in a var named container

var container = document.getElementById("wrapper");

//use Math.random to generate a random number

//between 0 and 1. Store random number in a var

var randomNumber = Math.random();

//use Math.floor() to round off the random number

//to an integer value between 0 and 10 and store it in a var

var numImages = Math.floor(randomNumber * 11);

//just for testing purposes, lets alert the resulting random number

alert(numImages);

//the loop will run for the number of times indicated

//by the resulting random number

for (var i=1; i <= numImages; i++)

{

//code to be executed:

//create a var and store a string made of a new HTML img element

//and the iteration variable i (use + concatenation operator):

//i contains the iteration number

//Take note:make sure the img src corresponds to a real image file

var newImage = '<img src="loop.gif" alt="#" />' + i;

//use the container var that stores the div element

//and use innerHTML to insert the string stored in the newImage var

//Use shortcut assignment += so previous values do not get wiped off

container.innerHTML += newImage;

}
```

```
</script>

</body>
</html>
```

Show example

Prepare a graphic file and store it in the same directory as your document, save your work, run the page in the browser and keep reloading it: the alert box will display a random number each time, then an img element is dynamically added to the web page as many times as indicated by the random number.

Questions, questions, questions

I know, there's a lot to ask about the code sample above. More specifically: 1) What's this **document.getElementById(element Id)**? 2) What are **Math.random() and Math.floor(number)**? And finally, 3) what's **innerHTML**? Let's tackle each question in turn:

1. **Get hold of HTML elements with** *document.getElementById(element Id)*

   JavaScript can perform all sorts of magic on any HTML element once it gets hold of it. This can be done in more than one way. However, the most straightforward way is by using the **document.getElementById(element Id)** command. The JavaScript interpreter reads it as follows:

   > "Hey browser, find the HTML element with the id name like the one in brackets."

   Because JavaScript usually does something with the element, this gets conveniently stored inside a variable. What you did in the sample code above was to grab hold of the div with the id attribute named *wrapper* and store it in a variable named *container*.

2. **Generate random numbers with Math.random() and Math.floor(number)**

   JavaScript has built-in commands to perform complex mathematical operations quickly and easily.

   Math.random() generates a random number between 0 and 1. Try it out on your own by typing *var randomNum = Math.random(); alert(randomNum);* inside a <script> tag.

Math.floor(number) rounds a number downwards to the nearest integer, and returns the result. The JavaScript interpreter reads it as follows:

> "Hey browser, take the number in brackets and round it down to the nearest integer number."

Try it out on your own by typing *var roundedNumber = Math.floor(6.2); alert(roundedNumber);* inside a <script> tag. Keep changing the number in brackets and see how JavaScript returns the corresponding integer number.

In the previous example, you used Math.floor() together with Math.random(number) to generate a random integer number between 0 and 10 (you typed 11 rather than 10 because the round number returned by Math.random() starts at 0 not 1).

3. **Add new mark-up inside HTML elements with** *innerHTML*

A JavaScript script can make HTML elements appear and disappear, change place or color, and much more.

There's more than one way to add new mark-up in your HTML document, but using **innerHTML** makes it very easy and straightforward. The JavaScript interpreter reads the command as follows:

> "Hey browser, take the **string to the right of the assignment operator** and insert it into the HTML mark-up."

What you did in the sample code above was to take the variable storing the wrapper div element and applying innerHTML to it to insert an img tag inside the div.

Try out: add images to the page with a *while loop*

Use the same HTML document from the previous example, but delete all the JavaScript between the <script> ... </script> tags. Now, type in the following JavaScript code:

```
<script type="text/javascript">
//Grab the html div by its id attribute
//and store it in a var named container
var container = document.getElementById("wrapper");
//create the var containing the counter
//and give it an initial value of 0
var numImages = 0;
//start the loop: while the number of images is less than or
```

```
//equal to 10 (loop starts at 0 not 1, so type 9 not 10),
//keep cycling and increase the counter by 1
while (numImages <= 9)
{
//this is the block of code to be executed:
//build the string to insert in the HTML document
//and store it in the newImage var
//Take note:make sure the img src corresponds to a real image file
var newImage = '<img src="loop.gif" alt="#" />' + numImages;
//use the container var that stores the div element
//and use innerHTML to insert the string stored in the newImage var
//Use shortcut assignment += so previous values do not get wiped off
container.innerHTML += newImage;
//increase the counter by 1 using the increment operator
numImages ++;
}
</script>
```

Show example

Make sure the image file is in place, save your work and run the page in the browser. You should see something similar to the example page indicated in the link above.

Run a while loop with a false condition

Now make just a small change to the code above: replace *var numImages = 0;* with *var numImages = 12;*. Now the **initial condition is false from the start**: the counter is a number greater than 10.

Save your work, run the page and see what happens. If all works fine, nothing at all should happen: the loop doesn't even get started.

Try out: add images to the page with a *do … while loop*

Use the same HTML document as the previous example. Also, the JavaScript code is very similar to the previous example: simply move the **while (condition)** outside the action block and type **do** in its place, like the example below:

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

```
var container = document.getElementById("wrapper");
var numImages = 0;
do
{
//this block gets executed first:
var newImage = '<img src="loop.gif" alt="#" />' + newImage;
container.innerHTML += newImage;
numImages ++;
}
//here's the condition to evaluate:
//the first cycle has already executed.
while (numImages <= 9);
</script>
```

Show example

Save your work and run the page in the browser. You should see something similar to the example page indicated in the link above.

Run a do ... while loop with a false condition

Make just a small change to the code above: once again, replace *var numImages = 0;* with *var numImages = 12;*. Now the **initial condition is false from the start**: the counter is a number greater than 10.

Save your work, run the page and see what happens. Unlike what happened with the *while loop*, now you should see one image displayed on the web page: the **loop has completed the first cycle and only afterwards the test condition is evaluated**. In this instance the condition is false and the loop stops **after the first iteration completes**.

Summary

That's all for this lesson. Your JavaScript toolbox is getting richer and richer: you can write repetitive code with loops, manipulate HTML content dynamically, and write scripts that generate random numbers.

Pat yourself on the back and take a well deserved break before moving on to the next big topic: **functions**.

Lesson 8: Package your JavaScript Code with Functions

By Maria Antonietta Perna

You can already add quite a bit of JavaScript functionality to web pages, and that's great. However, what if you decide to implement the same kind of functionality on more than one page, or in different places on the same page?

This is where **JavaScript functions** come into play.

In this lesson, you will learn:

- what **functions** are and how they are used in JavaScript;
- how to **retrieve information from functions**;
- how to **give information to functions**;
- what **variable scope** is all about.

In the process you will also learn how to:

- get today's date dynamically;
- change the value of an HTML button element dynamically.

Functions: what they are and what they are for

In previous lessons you used JavaScript functions to perform some actions quickly and easily like generating random numbers, writing something on a web page, etc. In fact, JavaScript has a great many built-in functions like **write(), alert(), getElementById(), random(), floor()**, and several others. Wherever you spot a pair of round brackets there's likely a function in action.

JavaScript also gives you the option to craft your own functions.

A function is a way of packaging your JavaScript commands so you can easily reuse them every time you need the same piece of functionality implemented in your website.

This suggests that there are 2 distinct phases to a function:

1. **The phase where the function is declared (created);**
2. **The phase where the function is called (used).**

Let's go through each phase in turn.

Declare a function

The basic structure of a JavaScript function looks like this:

```
//keyword function followed by your
//chosen name for the function
//Don't forget the round brackets!
function functionName()
//the code block is inside curly braces
{
//code you want to run goes here
}
```

Call a function

Once you put a block of code into a function, you can use it anywhere you need that functionality in your website. What you need to do is just **call the function**.

You do this by typing the function name followed by brackets and the function will do its job - just like you did with alert(). Here's how it's done:

```
//Type the function name (without the keyword function)
//Don't forget the round brackets!
functionName();
```

That's all you need: whatever code you stuffed into your function gets executed at this point.

Let's see how this works in practice. Fire off your text editor: it's time to get coding!

Try out: declare your function

Prepare a simple HTML page like the one below. The program we're going to build has the following goals:

- to get today's date dynamically when the user clicks a button;
- to display today's date on the web page;
- to change the value attribute of the button after it's clicked: to have the button still displaying *Get Date* after the date has been displayed looks a bit confusing to the user.

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 8: Declare a function</title>
</head>
<body>
<h1>Lesson 8: Declare a function</h1>

<div>

  <h2>Today's date is:</h2>

  <span id="calendar"></span>

  <input type="button" id="myButton" value="Get Date" />


</div>

<script type="text/javascript">

//Declare your function here

function showDate()

{

//the block of code starts here:

//First get all your vars ready

//This is how JavaScript retrieves today's date

var today = new Date();

//get hold of the calendar span element

//where today's date will be inserted

var myCalendar = document.getElementById("calendar");

//get hold of the button:you need this when it comes

//to change its value attribute

var myButton = document.getElementById("myButton");

//insert the date in the span element.

//toDateString() changes the date just retrieved

//into a user-friendly format for display

myCalendar.innerHTML = today.toDateString();

//change the value attribute of the button

//to say something more appropriate once the date is displayed

myButton.value = "Well done!";

}

</script>

</body>
</html>
```

Try out: call your function

If you review your program's goals as set out at the beginning, you'll see that **all the action takes place after the user clicks the button** on the page. This tells us that we need to **handle the button's onclick event**.

If you go back to lesson 3 for a moment, you remember that one way in which this can easily be done is to put some JavaScript code as the value of the **HTML element onclick attribute**.

Just add an *onclick* attribute to the button element and plug your showDate() function right in there, like so:

```
        <input type="button" id="myButton" value="Get Date"
onclick="showDate();" />
```

Show example

Save your work and run the page in the browser. Click the button and you should see today's date displayed on the page and the button value attribute giving you an appropriate message.

Questions, questions, questions

I know, there's some new stuff in the code samples above. You might be wondering: 1) what's this *new Date()* business? And 2) What does toDateString() do to the date? Let's tackle each question in turn.

1. ## What's new Date() all about?

   The *new* keyword creates a **new instance of a JavaScript object**, in this case a Date object based on the user's computer clock. Objects are the topic of the next lesson, and the Date object is the topic of lesson 11. Therefore, I keep things really short at this point.

   The important thing you need to know now is that once you've created an instance of the Date object, you've got all sorts of useful functions (called *the object's methods*) at your fingertips to manipulate date and time.

2. ## Why did I use *toDateString()* with the date?

   Following on from the previous point, **toDateString()** is only one of the numerous methods you can use with the Date object. The JavaScript interpreter reads it as follows:

"Hey browser, take the date you've been attached to and make it a bit more human-friendly!"

This is what I mean. Type the following inside enclosing <script> tags of an HTML page: *var myDate = new Date(); document.write(myDate);*

Save the page and run it in the browser: your date should be displayed in a format like the one below (obviously I expect the date to be different):

Sun Nov 06 2011 14:45:30 GMT+0000 (GMT Standard Time)

Ugly! Luckily, the Date object has its own beauty remedy. Rewrite the previous code snippet as follows: *var myDate = new Date(); document.write(myDate.toDateString());*

And here's what the date above looks like after its beauty treatment with *toDateString()*:

Sun Nov 06 2011.

Much better, less scary, and far more readable for us humans.

Feed information to and retrieve information from a function

Functions manipulate data: the alert() function has messages to display as its data, the write() function has text to write on the web page as its data, etc.

You can also **feed data to a function** for manipulation. The way you do this is through **arguments (or parameters)**. An **argument is placed inside the function's brackets when the function is declared**. You can place one or more arguments separated by commas(,) inside a function.

Here's what the basic structure looks like:

```
function functionName(arg1, arg2, arg3)
{
//body of the function goes here
}
```

The function argument is like a **placeholder for the value that gets fed when the function is called**. This will appear clearer in the example below.

Finally, one useful thing functions can do with data, once it's been manipulated, is to

return it.

For example, take the **floor(number) function** that you already know from the previous lesson. This function **takes in a number argument** and **returns an integer number**.

If your script does something with the **returned value then it needs to assign the function to a variable** when calling the function.

Let's put this knowledge into practice right away. Create a function and then call it to use its **return value** in your document.

Declare a function with a parameter

Prepare a simple HTML page and type the following JavaScript code within *<script> ... </script>* tags:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 8: Function Arguments and Return Values</title>
</head>
<body>
<h1>Lesson 8: Function Arguments and Return Values</h1>

<script type="text/javascript">

//declare a function with two arguments of type number

function addNumbers(num1, num2)

{

//this is a simple function:

// just return the sum of the two number arguments

return num1 + num2;

//your function ends here

}

</script>

</body>
</html>
```

Call your function and use its return value

Continue on from the previous code. Outside your function, just **after the } closing curly brace**, type the following:

```
//Call your function: create a var

//and assign the value returned by your function to it
```

```
//Also, give a value to the number arguments
//by typing 2 comma-separated numbers within brackets
var total = addNumbers(3, 5);
//Display the returned data on the page
document.write(total);
```

Save your work and run the page in the browser. If all goes well, the sum of the numbers you inserted as arguments of your addNumbers() function should be displayed on the web page.

Function arguments are not limited to a specific data type. You can also use strings and booleans, and even other functions. However, we don't want to be too involved at this stage.

Variables inside and outside functions: scope

A variable can have **local or global scope on the basis of whether it's declared inside or outside a function block.**. But, what does this mean exactly?

Simply put, a variable having **local scope means that it's visible, or accessible, only within the function in which it lives**. No other portion of your code can see a local variable.

On the other hand, a variable having **global scope means that it's accessible, that is, it can be used, anywhere in your script.**. Let's see this in practice. Get your text editor ready and type the following snippet between opening and closing <script> tags in an HTML page:

```
//create 2 global variables
var message = "outside global message";
var otherGlobalVariable = "other global variable";
//create a function with a local variable
//having the same name as the first global variable
function getMessage()
{
var message = "inside local variable";
//the function alerts the message variable
alert(message);
//and it also alerts the second global variable
alert(otherGlobalVariable);
//the function ends here
```

```
}
//call the function
getMessage();
//alert the message variable
//(which one will it be, local or global?)
alert(message);
```

Save your work and preview the page in your browser. As you can see, the **alert() function outside the variable doesn't have access to the message variable inside the function**. On the other hand, the **function can have access both to its local variable and to any variable in the global space**.

This might seem a bit confusing at first, but it's all a matter of practicing your coding skills as often as you can. The important thing to remember is that, if at times the variables seem not to have the values you expect, it might be a scope-related bug.

Summary

That's all for this lesson. You can now package your JavaScript code into reusable and manageable functions, input and output data via a function, use the Date object to retrieve and display the current date, change the value of an HTML button element on-the-fly, and distinguish between local and global variables. You've accomplished a great deal, congratulations!

Take a break and get ready for the next big topic, a core feature of contemporary programming languages, **objects**.

Lesson 9: A Gentle Introduction to Objects

By Maria Antonietta Perna

Object Oriented Programming (OOP) is a programming model used by most contemporary programming languages. JavaScript offers significant object oriented capabilities, has its own built-in objects, and offers you the option of creating your own custom objects.

This is a wide and complex subject and we're only going to scratch the surface in this tutorial series. However, because as soon as you start coding in JavaScript you can't avoid dealing with objects, it's important that you get familiar with some core concepts of OOP.

This short lesson is introductory to the more detailed and practical lessons on specific JavaScript objects that follow.

Here is what you will learn:

- the concept of *object* in web programming;
- what object **properties** are;
- what object **methods** are.

What is an object in Geeky talk?

Real world objects are things like books, cars, balls, etc. We humans deal with the world around us by interacting with and manipulating things, that is, objects.

What most computer programs do is manipulate, manage, and reuse data of various kinds. Therefore, using the real world object metaphor, contemporary programming languages like JavaScript deal with their virtual environment by populating it with packaged data modelled after the concept of an object.

Everything in JavaScript is an object and you've already used plenty of objects without realizing it. In fact, you're already familiar with:

- the **Document object** that you used to print your JavaScript messages on the web page;
- the **Math object**, that you used to generate random numbers and round off decimals;
- HTML elements like the **button object** that you used to manipulate its value attribute;
- the **Date object** that you used to retrieve the current date.

Properties and methods

But how exactly does JavaScript manipulate objects?

The answer is: not much differently from the way we humans manipulate real world objects. We do this by interacting with the qualities and capabilities that belong to individual objects.

Let's take a ball as our example. We interact with the ball by means of some of its qualities (its roundness, its hardness, its size, etc.). Also, we interact with the ball on the bais of what we expect the ball's behavior to be like (we expect that we can launch the ball without breaking it, that we can throw it against a wall and it bounces back, etc.).

Similarly, a JavaScript script interacts with its object-modelled data by means of the objects' qualities and behavior. These are called **properties and methods**.

Properties are values associated with an object. For example, an **HTML element object has a value property** (like the button object you're familiar with), and an **innerHTML property**, that you used to add new mark-up to the web page.

Methods represent what an object can do, its behavior, and are very much like functions.

How do you associate an object with a property or a method?

If you wondered what that odd-looking (.) dot notation in *document.write()* or *Math.random()*, and so on, meant in previous lessons, here's the answer.

You use the **object.property and object.method syntax** to interact with JavaScript objects.

```
//use the random() method of the Math object
Math.random();
//use the write() method of the document object
document.write();
//use the length property of the string object
myStringText.length;
```

Summary

This lesson has been a short introduction to the concept of JavaScript **objects, their properties and methods**.

In the next lesson you will learn about the most widely used properties and methods of the **String object**.

Lesson 10: JavaScript Objects - Strings

By Maria Antonietta Perna

In the previous lesson you were introduced to the concept of **Object Oriented Programming (OOP)**. You're now familiar with the use of objects and their **properties and methods** in JavaScript programming.

Starting from this lesson and for the next 3 lessons, we will be *getting up close and personal* with some of the most common properties and methods of widely used JavaScript objects: **String, Date, Math, and Array**. However, because JavaScript as a programming language is mostly made of objects, you will keep learning to use more objects and related properties and methods throughout the rest of this tutorial.

In this lesson you will learn how to use the following **properties and methods of the string object**:

- **length property**;
- **toLowerCase() / toUpperCase()**;
- **match()**;
- **replace()**;
- **indexOf()**.

In the process, you will also **practice putting your JavaScript code into an external file**.

The String object

You've had plenty of practice with the **string object** through this tutorial. In fact, the JavaScript interpreter reads any piece of text enclosed in quotes ' ' (or double quotes " ") as an instance of the string object. This puts the magic of all the properties and methods belonging to the string object in our hands.

The beauty of **Object Oriented Programming** is that we can use all that JavaScript goodness to achieve our script's goals without needing to have any clue whatsoever of the inner workings of those properties and methods. All we need to know is **what a property or method can do** for us, not **how it does** it.

For example, if we need to know the length of a piece of text, just using **pieceOfText.length** will achieve this. This is accomplished without us knowing anything of the programming virtuosity responsible for the power of the **length property of the string object**.

Let's have a taste of such power.

How to use *length*

The **length property of the string object** contains the number of characters (including spaces) in the text value of a string.

Here's a basic code snippet to demonstrate its use:

```
//create and initialize a string variable
var myString = "Hello JavaScript"
//apply the length property to the string
document.write(myString.length);
//JavaScript will print 16:
//spaces are included in the count
//If the string is empty length returns 0
```

Try out for yourself: insert the code above between enclosing <script> tags and have fun experimenting with it.

How to use *toUpperCase()*

The **toUpperCase() method of the string object** turns the text value of a string into, well ... uppercase letters. Its companion **toLowerCase()** does the opposite: it turns the text value of a string into lowercase letters.

Here's a basic code snippet to demonstrate its use:

```
//create and initialize a string variable
var myString = "Hello JavaScript"
//apply the toUpperCase() method to the string
document.write(myString.toUpperCase());
//JavaScript will print HELLO JAVASCRIPT
//Try using myString.toLowerCase() on your own
```

Have a try: insert the code above between enclosing <script> tags and switch uppercase into lowercase.

How to use *match()*

The **match() method of the string object** is used to search for a specific value inside a string.

Here's a basic code snippet to demonstrate its use:

```
//create and initialize a string variable
var myString = "Hello JavaScript"
//apply the match() method to the string.
//match(stringToMatch) takes the value to search for as argument
document.write(myString.match("JavaScript"));
//JavaScript will print JavaScript
//If no match is found the method returns null
```

Experiment on your own: insert the code above between enclosing <script> tags and try matching different string values.

How to use *replace()*

The **replace() method of the string object** is used to replace a value for another value inside a string.

Here's a basic code snippet to demonstrate its use:

```
//create and initialize a string variable
var myString = "Hello JavaScript"
//apply the replace() method to the string.
//replace() takes 2 arguments and returns the new string value:
//replace(valueToReplace, newValue)
document.write(myString.replace("JavaScript", "World"));
//JavaScript will print Hello World
```

Have a go: insert the code above between enclosing <script> tags and try replacing different string values.

How to use *indexOf()*

The **indexOf() method of the string object** is used to know the position of the first found occurrence of a value inside a string.

Here's a basic code snippet to demonstrate its use:

```
//create and initialize a string variable
```

```
var myString = "Hello JavaScript"
//apply the indexOf() method to the string.
//indexOf() takes in the value to look for in the string as argument
//and returns the position number (index)
//indexOf(valueToFind)
document.write(myString.indexOf("Java"));
//JavaScript will print 6
//indexOf() includes spaces and starts counting at 0
//if no value is found the method returns -1
```

Have a go on your own: insert the code above between enclosing <script> tags and print the index position of different letters inside the string.

Try out: guessing game

You're going to build a simple guessing game application using all the new knowledge you've acquired in this lesson. Your application is expected to check a name entered by the user and respond accordingly:

Your **JavaScript code will be placed in its own external file** and referenced from the HTML document. **Separation of concerns**, that is, **separation between structure (HTML mark-up), appearance (CSS), and behavior (JavaScript)**, reflects contemporary best practices in web design.

Let's start with the HTML document. This is a simple web page with a textbox, a button, and a div where the JavaScript result will be displayed:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 10: JavaScript Objects - Strings</title>
<script type="text/javascript" src="lesson10.js"></script>
</head>
<body>
<h1>Lesson 10: JavaScript Objects - Strings</h1>

<h2>What's the name of the wizard boy in J.K. Rowling's novels?</h2>

<p>Your answer: <input type="text" id="txtName" /></p>
<p><input type="button" value="Submit your answer" id="btnAnswer" />

<p id="message"></p>

</body>
</html>
```

Pay attention to the enclosing <head> tags. These contain a **reference to an**

external JavaScript file named *lesson10.js* located in the same directory as the HTML file (you're free to place the JavaScript file in its own directory, if you prefer. However, make sure this is reflected in the src value of your <script> tag in the HTML document).

Also, take note of the fact that the **elements that are going to play a role in our script all have an** *id attribute***. This gives JavaScript a hook on those elements** and the data they will eventually contain.

Now open the JavaScript file *lesson10.js* and type the following code:

```
//This is the function that initializes our program
function init()
{
//Store a reference to the HTML button element in a variable:
//use the id of the HTML button element to do this
var myButton = document.getElementById("btnAnswer");

//use the onclick event of the button
//and assign the value of the getAnswer() function to it.
//This function performs the main job in your application
//and runs after the user clicks the button on the page.
//Take note: getAnswer is assigned without brackets.
//This is so because otherwise getAnswer() would be called
      //as soon as the page loads (and we don't want that).
myButton.onclick = getAnswer;

//the init function ends here
}

//Assign the init() function to the onload event:
//this event fires when the HTML page is loaded in the browser.
//Take note: init is assigned without brackets
onload = init;

//Now write the getAnswer() function
function getAnswer()
{
//Create all the vars you need to manipulate your data:
//secretName stores the correct answer the user is expected to guess:
var secretName = "Harry Potter";
```

```
        //Turn the value of secretName into lower case:
        //you do this because you're going to compare this value
        //to the value entered by the user of your application.
        //Given that users might type the answer either in upper or lower case,
        //reducing the relevant text values to the same casing automatically
        //ensures that only the content and not also the letter case plays a
role in the comparison.
        var secretNameLower = secretName.toLowerCase();

        //Get the value the user types into the textbox
        var myTextBox = document.getElementById("txtName");
        var name = myTextBox.value;

        //Also turn the value entered by the user to lower case
        var nameLower = name.toLowerCase();

        //Get a reference to the HTML paragraph that will display your result
        //after the script has run by storing its id value in a var
        var message = document.getElementById("message");

        //These are the test cases your application needs to evaluate
        //and respond to: if the user clicks the button but did not
        //enter any value in the textbox:
        if(nameLower.length <= 0)
        {
        alert("I didn't quite catch your answer.  Please enter an answer");
        }

        //If the user gets right the first half but not the latter half of the
name:
        //Take note of the use of the logical operator &&
        //(go back to lesson 5 if you need to revise logical operators)
        else if(nameLower.indexOf("harry") == 0 && nameLower.indexOf("potter")
== -1)
        {
        alert("Almost there: think again");
        }

        //If the secret name and the name entered by the user match:
        else if(nameLower.match(secretNameLower))
        {
```

```
alert("You got it!");
message.innerHTML = "Congratulations, you win!";
}

//Default case - if the user types in the wrong answer:
else
{
alert("Wrong!");
message.innerHTML = "Sorry.  The correct answer is: ";
message.innerHTML += name.replace(name, "Harry Potter");
}

//the getAnswer() function ends here
}
```

Show example

Save all your files and run the HTML page in the browser. You should see something similar to the example page indicated in the link above.

If you **click the button without entering any value** in the textbox, an alert will pop up inviting you to enter an answer.

If you **get right the first half ("Harry") - indexOf() returns 0 ("Harry" is at position 0, which is the start of the index) - but not the latter half ("Potter") - indexOf() returns -1 (it finds no corresponding value inside the string)**, you get an alert letting you know that your answer is almost right.

If you **enter the correct answer**, well you'll get a prize.

Finally, if your **answer is totally wrong**, your application will say so and give you the correct answer.

The above code sample employs all the JavaScript tools you learned in this lesson and most of those you encountered in previous lessons. Experiment with it as much as you can. If you have doubts, leave a message in our JavaScript forum.

The really new stuff in the guessing game application is the way JavaScript functions work from an external file. This is the technique I'm going to use for our *Try out exercises* in most of the remaining part of this tutorial - at least until jQuery makes its appearance. This means that you'll be getting plenty of practice with it.

Summary

In this lesson you started to have a taste of the power of objects. You also started to use an external JavaScript file for your *Try out exercise*, thereby conforming to best practice in contemporary web design techniques. Now have a break and get yourself ready for the next topic: the **Date object**.

Lesson 11: JavaScript Objects - Date Object

By Maria Antonietta Perna

In the previous lesson, you had a go at manipulating text by taking advantage of the **powerful properties and methods of the String object**.

Here we continue to explore the power of JavaScript objects: we will be looking into the magical virtues of the **Date object**.

Here's what you will do in this lesson:

- **create a Date object**;
- **set dates**;
- **compare dates**.

The Date object

In lesson 8 you learned how to retrieve the current date and display it on the web page in user-friendly format. This could be done dynamically thanks to the JavaScript goodness offered by the Date object.

The **Date object** exposes plenty of methods to manipulate dates and times, some of which we're going to examine here.

I **recommend you try out all the demos below** by inserting the code snippets between enclosing <script> tags in an HTML page (for such short demos we can get away with not using an external JavaScript file). Also, feel free to **experiment with each demo as much as possible** before moving on to the next example.

Create a Date object

You create a date object by using one of the following ways:

```
//First way:
//the var today is initialized with a Date object
//containing the current date and time
//on the basis of the user's computer
//use toLocaleDateString() to adjust the date
//to the local time zone and format it automatically
var today = new Date().toLocaleDateString();
```

```
                    //Second way:
                    //you pass a millisecond argument that starts at
1970/01/01:
                    //new Date(milliseconds)
                    var date = new Date(1000).toLocaleDateString();
                    //On my system this outputs: 01 January 1970


                    //Third way:
                    //You pass a string as argument:
                    //new Date(dateString)
                    var date = new Date("10 November,
2011").toLocaleDateString();
                    //On my system this outputs: 10 November 2011


                    Fourth way:
                    //new Date(year, month, day, hours, minutes, seconds,
milliseconds)
                    //only year and month are required,
                    //the others are optional arguments and
                    //where not present 0 is passed by default
                    //Below you pass the year in a 4-digit format, and the
month in number form:
                    //months are represented by numbers from 0 (January) to 11
(December)
                    var date = new Date(2011, 10).toLocaleDateString();
                    //On my system this outputs: 01 November 2011
```

After a Date object is created, you can use its methods to get and set dates and times.

Use getDate() to retrieve a date

```
//Create a Date object containing the current date and time
var myDate = new Date();
//use getDate() to extract the  day of the month
document.write(myDate.getDate());
//At the time of writing, this prints 11
//days of the month range from 1 to 31
```

Use getTime() to retrieve a time

```
//Create a Date object containing the current date and time
var myTime = new Date();
//use getTime() to extract the  time
document.write(myTime.getTime());
//At the time of writing, this prints 1321021815555
//This is the millisecond representation of the current
//date object: the number of milliseconds between
//1/1/1970 (GMT) and the current Date object
```

Get Date object components

Once you have a Date object, one interesting thing you can do with it is to get its various components. JavaScript offers some interesting methods to do just that.

Use getFullYear() to extract the year component

```
//Create a Date object containing the current date and time
var myDate = new Date();
//extract the year component and print it on the page
document.write(myDate.getFullYear());
//At the time of writing, this prints 2011
```

Use getMonth() to extract the month component

```
//Create a Date object containing the current date and time
var myDate = new Date();
//extract the month component and print it on the page
document.write(myDate.getMonth());
//At the time of writing, this prints 10
//which represents the month of November - months are represented
//with numbers starting at 0 (January) and ending with 11
(December)
```

Use getDay() to extract the day component

```
//Create a Date object containing the current date and time
var myDate = new Date();
//extract the day component and print it on the page
document.write(myDate.getDay());
//At the time of writing, this prints 5
//which represents Friday - days are represented
//with numbers starting at 0 (Sunday) and ending with 6 (Saturday)
```

Use getHours() and getMinutes() to extract time components

```
//Create a Date object containing the current date and time
var myDate = new Date();
//extract the hours component
var hours = myDate.getHours();
//extract the minutes component
var minutes = myDate.getMinutes();
//format and display the result on the page:
//check the minutes value is greater than 1 digit
//by being less than 10
if (minutes < 10)
{
//If it's just a one digit number, then add a 0 in front
minutes = "0" + minutes;
}
var timeString = hours + " : " + minutes;
document.write(timeString);
//Based on my computer clock, this prints:
//14 : 44
```

Set dates

You can also set dates and times as easily as calling the appropriate methods. Here are a few examples.

Use setFullYear() to set a specific date

In this demo, you will set a specific date and then retrieve its day component for display

```
//Create a Date object containing the current date and time
var myDate = new Date();
//set the Date object to a specific date: 31 October, 2011
var mySetDate = myDate.setFullYear(2011, 9, 31);
//Now retrieve the newly set date
var mySetDay = myDate.getDay(mySetDate);
//mySetDay will contain a number between 0 - 6 (Sunday - Saturday).
//Use mySetDay as test case for a switch statement
//to assign the corresponding week day to the number value
switch (mySetDay)
{
case 0:
mySetDay = "Sunday";
break;
case 1:
mySetDay = "Monday";
break;
case 2:
mySetDay = "Tuesday";
break;
case 3:
mySetDay = "Wednesday";
break;
case 4:
mySetDay = "Thursday";
break;
case 5:
mySetDay = "Friday";
break;
case 6:
mySetDay = "Saturday";
break;
}
//display the result on the page
document.write("The 31st October 2011 is a " + mySetDay);
```

```
//If you check the date on a calendar, you'll find that
//the 31st Oct 2011 is indeed a Monday
```

Compare 2 dates

In this demo you will compare 2 Date objects.

```
//Create a Date object
var myDate = new Date();
//set the Date object to a specific date: 31 October, 2011
var mySetDate = myDate.setFullYear(2011, 9, 31);
//Create a second Date object
var now = new Date();
//Make the comparison: if the set date is bigger than today's date
if (mySetDate > now);
{
//the set date is in the future
document.write("The 31st October is in the future");
}
//if the set date is smaller than today's date, it's in the past
else
{
document.write("The 31st October is in the past");
}
```

Summary

In this lesson you looked into some of the methods of the **Date object** and had the chance to experiment with them.

In the next lesson you will be exploring JavaScript mathematical wizardry: get ready for the **Math object**.

Lesson 12: JavaScript Objects - Math Object

By Maria Antonietta Perna

We continue on our journey through JavaScript built-in objects by offering a quick overview of the **Math object**.

You already had occasion to taste the power of the Math object back in lesson 7, where you used its **random() method** to generate a random number between 0 and 1, and its **floor() method** to round down the resulting random number.

The **Math object** allows you to write scripts that perform complex mathematical tasks in the blink of an eye.

In this lesson you will use:

- **Math.PI** to calculate the circumference of a circle;
- **Math.sqrt()** to calculate a square root value.

In the process, you will learn how to use:

- **parseInt()/parseFloat()** to convert strings to numbers;
- **isNaN()** to check whether a value is or is not a number;

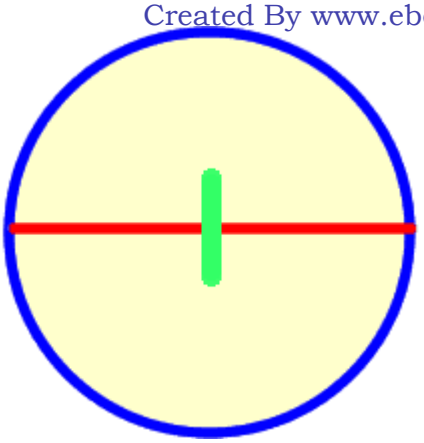
Math.PI

This property stores the value of PI, a mathematical constant that amounts to approximately 3.14159.

Let's quickly refresh our school math. Here are some definitions:

Circle

A circle is a shape with all points the same distance from the center:



- the round border surrounding the shape (the color blue in the graphic above), is the circle **circumference**;
- the line cutting the circle horizontally in half (the color red in the graphic above) is the circle **diameter**;
- each half of the diameter represents a **radius**.

PI is the **ratio of the circumference of a circle to the diameter**.

If you need this value in your JavaScript program, use **Math.PI**. Just try this one out between enclosing <script> tags of an HTML page:

```
//store and display the value of PI:
var pi = Math.PI;
document.write(pi);
```

Try out: use Math.PI to find out the circumference of a circle

It's time for this lesson's first *try out exercise*. We're going to build a simple application where the user enters a number representing the radius of a circle, clicks a button, and the program calculates the circumference and displays it on the page.

Keeping in mind that the radius is half the diameter, and that the formula to calculate the circumference is **PI * diameter**, our JavaScript statement to calculate the circumference given the radius will be: **2 * (Math.PI * radius)**.

First things first: prepare an HTML document like this one:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 12: JavaScript Objects - Math Object</title>
<script type="text/javascript" src="lesson12_tryout1.js"></script>
</head>
<body>
<h1>Lesson 12: Circumference calculator</h1>
```

```
<p>Enter length of radius: <input type="text" id="txtRadius"
/></p>

<p><input type="button" id="btnSubmit" value="Calculate
circumference" /></p>

<p id="result"></p>

</body>
</html>
```

The HTML page above references an external JavaScript file, *lesson12_tryout1.js*, and contains an inputbox, a button, and a paragraph where the result of the calculation will be displayed. Make sure these 3 crucial elements have an **id value** so your JavaScript script will have an easy to reach hook on those elements.

Now create a JavaScript document and name it *lesson12_tryout1.js*. This file will contain 3 functions:

1. **init()** initializes the script by binding the *processAnswer() function* to its appropriate event, that is, **the btnSubmit onclick event**;
2. **processAnswer()** checks the answer submitted by the user, calls the function that performs the calculation, and displays the result;
3. **getCircumference(rad)** gets passed the radius value as argument, performs the actual calculation, and returns the circumference.

```
//init() function binds the processAnswer() function
//to the onclick event of the button
function init()
{
var myButton = document.getElementById("btnSubmit");
myButton.onclick = processAnswer;
}

/*****************************************/

//Bind the init function to the onload event
onload = init;

/*****************************************/

//This function checks the user's input,
//calls the function that performs the calculation,
//and displays the result
function processAnswer()
{
```

```
//store a reference to the inputbox
var inputBox = document.getElementById("txtRadius");
//get the radius value entered in the inputbox
//make sure to convert the string (text) to a number:
//calculations can only be performed with numbers,
//and the value in the inputbox is processed as string.
//parseInt(string) and parseFloat(string)
//take a string as input and return an integer
//or a floating point number respectively.
var radius = parseInt(inputBox.value);
//get a reference to the paragraph to display result
var result = document.getElementById("result");
//create and initialize a var to store the result
var resultString = "";
//Perform some validation on the user's input:
//if the value entered by the user is not a number
//alert the user.  isNaN(value) takes
//a value as input and returns true if the value is not a number,
//it returns false if the value is a number.
//the expression below is short for:
//if (isNaN(radius) == true)
if (isNaN(radius))
{
alert("Please, enter a number");
}
else
{
//if the user has correctly entered a number,
//call the function that performs the calculation
//and display the result on the page
var circ = getCircumference(radius);
resultString = "Circumference is: " + circ;
result.innerHTML = resultString;
}
}

/*******************************************/
```

```
//This is the function that performs the calculation:
//it takes the radius as input and returns the circumference
function getCircumference(rad)
{
var circumference = 2 * (Math.PI * rad);
return circumference;
}
```

Show example

Save all your files and run the HTML page in the browser. You should see something like the page indicated by following the example link above.

If you click the button without entering any value in the inputbox, or if you enter a letter rather than a number, the program asks you to enter a number value.

As you click the button, the calculation is performed and the result is displayed on the page.

Math.sqrt()

Math.sqrt(number) takes a **number as argument and returns the square root of that number**.

If a negative number is entered as argument, for instance **Math.sqrt(-5)**, the function returns **NaN**, that is, a value that JavaScript does not treat as a number.

Brushing up on our school math, the square root of 25, for example, is that number that multiplied by itself yields 25 as a result, that is, 5.

The formula is: **square of n = n * n**. JavaScript performs this calculation automatically. Here's a quick demo: type the following code between enclosing <script> tags of an HTML page:

```
var squareRoot = Math.sqrt(25);
document.write("The square root of 25 is: " + squareRoot);
//This should print:
//The square root of 25 is: 5
```

Try out: square root calculator

You will build a simple square root calculator. Here are the requirements for this little application:

- the user will be able to enter a number value into an inputbox;
- the user will be able to click a button that calculates the square root of the number entered into the inputbox;
- the result will be displayed on the web page.

Create a fresh HTML page with an inputbox, 1 button, and a paragraph where the result will be displayed. These HTML elements contain **id attributes** that will provide a handy hook for our JavaScript code.

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 12: JavaScript Objects - Math Object</title>
<script type="text/javascript" src="lesson12_tryout2.js"></script>
</head>
<body>
<h1>Lesson 12: Square root calculator</h1>

<p>Enter a number: <input type="text" id="txtNumber" /></p>

<p><input type="button" id="btnSubmit" value="Calculate square
root" /></p>

<p id="result"></p>

</body>
</html>
```

Now create the *lesson12_tryout2.js* file. This file will contain 3 functions:

1. **init()** initializes the script by binding the *displaySquare() function* to its appropriate event, that is, **the btnSubmit onclick event**;
2. **displaySquare()** checks the answer submitted by the user, calls the function that performs the calculation, and displays the result;
3. **calculateSquare(input)** gets passed the number value entered by the user as argument, performs the actual calculation, and returns the the square root.

```
//write the function to initialize the script

function init()

{

//Bind function that processes the result to

//the onclick event of the button

var myButton = document.getElementById("btnSubmit");

myButton.onclick = displaySquare;

}
```

```
/*************************************************/

//Bind the init() function to the onload event
onload = init;

/*************************************************/

//Here we call the function that performs
//the square root calculation, then we format and
//display the result
function displaySquare()
{
//we store the value from the inputbox into a var.
//Notice how we concatenate several methods
//in one statement (be careful with those brackets).
//If you find it a bit confusing, store a reference to the
inputbox
//in its own var and then use it to extract its value property
//and finally pass it as argument of parseInt(), like so:
//var inputBox = document.getElementById("txtNumber");
//var inputVal = parseInt(inputBox.value);
var inputVal =
parseInt(document.getElementById("txtNumber").value);
//we store a reference to the paragraph
//where the result will be displayed
var result = document.getElementById("result");
//we create the var that stores the result message
var resultMessage = "";
//we ensure the input value is a number:
//if the user did not enter a number, we send an alert
if (isNaN(inputVal))
{
alert("Please, enter a number");
}
else
{
//If the input is in correct format, we call
//the function that performs the calculation
```

```
            var squareVal = calculateSquare(inputVal);
            //this is a nested if statement: it's inside another
            //if statement to perform further checks:
            //if squareVal stores a value (if the calculation was successful)
-
            //- this is short for: if (squareVal == true)
            if (squareVal)
            {
            //we build this message to the user:
            resultMessage = "Square root of " + inputVal + " is " +
squareVal;
            }
            //else, that is, if the calculation was not successful
            else
            {
            //we build a different message to the user:
            resultMessage = "Sorry, an error occurred";
            }
            }
            //finally, we display the message using innerHTML
            result.innerHTML = resultMessage;
            }

            /************************************************/

            //This function calculates the square root:
            //it takes in the number entered by the user
            //and returns the result of the calculation
            function calculateSquare(input)
            {
            var squareVal = Math.sqrt(input);
            return squareVal;
            }
```

Show example

Save all files and preview your work in a browser. You should see something similar to the example indicated by following the link above.

Enter a number into the inputbox and click the button. If you enter anything but a number into the inputbox (or if you leave the inputbox empty), you'll be alerted by a popup box asking you to enter a number. If all goes well, the square root of the number you enter in the inputbox will be displayed on the page.

If the application is not working, check your typing, especially letter casing, brackets, and semi-colons (;). If you have any doubts, just drop us a line in the forum dedicated to this tutorial.

Summary

You've made it to the bottom of the page, congratulations! After this lesson you've added new tools to your JavaScript toolbox: the **PI property and sqrt() method of the Math object**. You also know how to convert a string (text) to a number data type with **parseInt() and parseFloat()**, so that you can use the value to perform calculations. Finally, you learned how to check user input with **isNaN()**.

It's time for your break before the next big topic: the **Array object**.

Lesson 13: JavaScript Objects - Arrays

By [Maria Antonietta Perna]

Just using a few properties and methods of common JavaScript objects has opened your eyes to the amazing things that **Object Oriented Programming** enables you to accomplish with a few lines of code. In this lesson, you will learn to use what is a very powerful object in most programming languages, including JavaScript of course: the **Array object**.

In particular, in this lesson you will learn:

- what the **Array object is for**;
- how to **create arrays**;
- how to **access, sort, add, and remove data inside an array**.

In the process, you will learn how to use the **for ... in loop** to access data inside an array.

What is the Array object for?

An object's job is that of storing and allowing for the easy manipulation of data. We can safely say that the **Array object** performs both tasks wonderfully.

From the perspective of data storage, an array works a bit like a variable. The difference is that you are not limited to one piece of data, as it's the case with common variables. You can stuff several items at once inside an array and retrieve whichever you need at the right time in your program.

Imagine if you had to have a storage place for each pair of socks. This would be a nightmare! Having a drawer to store all your socks in one place makes much more sense. In this respect, arrays are a bit like drawers, but much more powerful.

From the point of view of data manipulation, the Array object offers a wealth of properties and methods, some of which you will examine in this lesson.

How do you create an Array object?

You create an array in 1 of 3 ways:

1) use *new Array()*

This is the most verbose approach. It works as follows:


```
var colors = new Array();
colors[0] = "green";
colors[1] = "red";
```

2) use *new Array(item0, item1, item2 ...)*

This approach also employs the *new Array()* constructor, but does so using a more concise formulation. Here's what it looks like:

```
var colors = new Array("green", "red");
```

3) use the *literal array* approach

This is the shortest way of creating and populating an array, and it's the most widely used approach. It looks like this:

```
var colors = ["green", "red"];
```

All 3 approaches illustrated above, create an array object called *colors* containing 2 string values, "green" and "red" - but arrays can store all other data types: numbers, booleans, even other arrays! The items in our array occupy position *0* and *1* respectively.

As you might have guessed, you **count the index position of an item inside an array starting at 0**.

But, is this index position important? Well, yes it is, at least if you want to use the items inside the array, which is most likely the case.

Here's what I mean.

How do I access data inside the Array object?

Arrays are great to store several values using only 1 variable. However, they wouldn't be so great if you couldn't also access those values quickly and easily. Here's how it's done:

```
//"green" is at index position 0, "red" is at index position 1, etc.
var colors = ["green", "red", "yellow", "orange", "blue"];
var red = colors[1];
```

```
document.write(red);
```

Enter the code snippet above between enclosing <script> tags in an HTML page, save your work, and run the page in the browser. You should see the word *red* displayed on the page.

What you did was to access the second value in the *colors array* by using its index position within the array starting to count at 0.

Access all items in the array with a loop

You might have probably guessed this, but loops and arrays go together like cookies and milk. Here's how you would access all values inside an array by looping over each of them:

```
//use the array from the previous example
var colors = ["green", "red", "yellow", "orange", "blue"];
//use a for loop to access each item in colors
//The array object has a length property
//that contains the size of the array: you can use it in the for
loop

for (var i=0; i < colors.length; i++)
{
//You use the counter i to access the index position
document.write(colors[i] + "<br />");
}
```

Save your file and preview the page in a browser. You should see the color names displayed on the page.

There's also a special type of for loop that you can use with the Array object, the **for ... in loop**. Here's how it works:

```
//use the array from the previous example
var colors = ["green", "red", "yellow", "orange", "blue"];
for (color in colors)
{
//color is the variable that marks the index position
document.write(colors[color] + "<br />");
}
```

Run the HTML document containing the code snippet above inside <script> tags in a browser. The web page should look the same as it did in the previous example.

Once you access a value inside an array you can simply retrieve it, as you did in the previous example, or you can modify it, as follows:

```
var colors = ["green", "red", "yellow", "orange", "blue"];
colors[2] = "pink";
//Now you've replaced the item at index position 2,
//the third item called "yellow",
//with "pink"
document.write(colors);
//The code should print:
//green,red,pink,orange,blue ("pink" has replaced "yellow")
```

Can I sort array items?

You can do amazing stuff with values stored inside an array. For example, you can sort items alphabetically, in descending or ascending order, or numerically. Let's see how this is done.

You sort array elements with a fantastic method most appropriately called **sort()**. Here it is in action:

```
//sort the colors array alphabetically and ascending:
var colors = ["green", "red", "yellow", "orange", "blue"];
var ascendingColors = colors.sort();
//display each value
document.write(ascendingColors);
//This should print:
//blue, green, orange, red, yellow
```

If you want to **sort numerical values inside an array, either in ascending or descending order**, you need to build a simple **custom function and pass this function as argument to the sort() method**. It sounds a bit harder than it actually is. Here's how it's done:

```
//build a new array containing number values
```

```
var myNumbers = [4, 2, 10, 6, 9];
//build your custom function: this is a simple formula
function sortAscending(a, b)
{
//the formula return a - b sorts the numbers
//from the smaller to the bigger,
//the formula return b - a sorts numbers in descending order,
//that is, from the bigger to the smaller number
return a - b;
}
//pass the sortAscending function as argument to sort()
var sortedAscending = myNumbers.sort(sortAscending);
//print result on the page
document.write(sortedAscending);
//This should print:
//2,4,6,9,10 .  Now try sorting in descending order on your own.
```

How do I add data to the Array object?

You can add new items inside an array in different ways. Which method you choose depends largely on your program's requirements.

Add items to the end of the array with push()

What feels more like the natural order of things, that is, adding an item to the end of an array, is easily achieved using the **push() method**. Here's how it works:

```
//use our old colors array
var colors = ["green", "red", "yellow", "orange", "blue"];
//use push() to add the color "pink" at the end
colors.push("pink");
//print the enlarged array
document.write(colors);
//This should print:
//green,red,yellow,orange,blue,pink
```

Add items to the beginning of an array with unshift()

Use **unshift()** to add new items to the start index position of an array. This is easily done.

Just replace *push()* with *unshift()* in the previous example, save your work and run the page in a browser. The page should display the color name "pink" at the very start, like so:

pink,green,red,yellow,orange,blue.

How do I remove data from the Array object?

If you want to remove the last item in an array you use **pop()**. If you want to remove the first item in an array you use **shift()**.

```
var colors = ["green", "red", "yellow", "orange", "blue"];
//remove "blue" from colors
colors.pop();
document.write(colors);
//This prints: green,red,yellow,orange


//remove "green" from colors
colors.shift();
document.write("<br />" + colors);
//This prints: red,yellow,orange
//on a new line break
```

Have fun experimenting with the examples above and come back to our *try out* challenge as soon as you're ready.

Try out: Username registration

You will build a mock Username registration program with the following requirements:

- the user can **enter a Username in the inputbox**, and click a button to register;
- if the Username already exists, the program informs the user;
- if the Username is not already stored in the program, the new Username is added, the user is informed, and all stored Usernames are displayed on the page in alphabetical order.

A real registration program would ask for more information, the data would be permanently stored in a database, and existing Usernames would never be printed out for everybody to see.

However, this is a mock application: our Usernames are stored in a **global array**. This ensures that the values inside will be accessible from any function in the script, and the newly added Usernames will be preserved for as long as the program runs. Displaying the Usernames on the page will show the workings of the **global variable (see Lesson 4 for a refresher on global variables)** as well as of the array methods **push()** and **sort()**.

Enough talking, let's get coding! Prepare an HTML page with an inputbox, a button, and a paragraph to display messages to the user. Make sure each element has an **id value** so that your code can have an easy handle on them.

The HTML document

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 13: JavaScript Objects - Arrays</title>
<script type="text/javascript" src="lesson13_tryout.js"></script>
</head>
<body>
<h1>Lesson 13: Register with us</h1>

<p>Register a Username: <input type="text" id="txtName" /></p>

<p><input type="button" id="btnSubmit" value="Register" /></p>

<p id="result"></p>

</body>
</html>
```

The JavaScript file: *lesson13_tryout.js*

Your program contains:

- a **global array called userNames** used to store Usernames;
- a function called **init() used to bind the main function to the onclick event of the button**;
- and a function called **registerName()** that performs all the main program tasks.

```
//global array: it's outside any function

var userNames = ["harry potter", "donald duck", "shrek",
"supercoder"];

/*******************************/

//init() function

function init()

{
```

```
var myButton = document.getElementById("btnSubmit");
myButton.onclick = registerName;
}

//assign init() function to onload event
onload = init;

/*****************************************/

//registerName() function: it executes when user clicks the button
function registerName()
{
//set up main vars: Username entered by user,
//a message string to communicate with the user,
//a reference to the paragraph used to display the message,
//and a boolean var (true/false) used as flag:
//if the registration is successful, this is set to true,
//if registration fails, this is set to false. It's initialized as
false.
//Notice how you chain getElementById(), value, and toLowerCase
//to store the value entered by the user in lowercase
var newName =
document.getElementById("txtName").value.toLowerCase();
var message = "";
var result = document.getElementById("result");
var success = false;
//If the user clicks the button but the inputbox is empty
//we alert the user and stop further program execution:
if (newName == "")
{
alert("Please, enter a Username");
return false;
}
//we loop over each Username stored in the array
//to make sure the Username is not already in existence
for (var i = 0; i < userNames.length; i++)
{
//if we find a Username equal to the newly entered name,
//it means the Username already exists and we can't
```

```
                //proceed with registration
                if (userNames[i] == newName)
                {
                message = "Sorry, the Username " + userNames[i] + " already
exists.  Try again";
                result.innerHTML = message;
                //set the success flag to false so the rest of the program knows
                //that registration failed
                success = false;
                //stop further program execution
                return false;
                }
                //else - if the Username entered by the user
                //is not already stored in the application, register new user:
                else
                {
                message = "Great, you've successfully registered with us as " +
newName;
                result.innerHTML = message;
                //set the success flag to true, so the program knows
                //the new Username can be added
                success = true;
                }
                }
                //Now you're out of the loop
                //if registration was successful (success flag is true),
                //add new Username to the array with push()
                if (success)
                {
                userNames.push(newName);
                }
                //display Usernames sorted alphabetically on a new line
                result.innerHTML += "<br />" + userNames.sort();
                }
```

Show example

Save all your files and run the HTML document in a browser. You should see something

similar to the example page indicated by following the link above.

Try clicking the button without entering anything in the inputbox, then enter a Username already present in the array - for example, supercoder - and try to register; finally, enter a new name and see it getting added to the array. As you enter new names, these get added to the array and displayed alphabetically.

Summary

You've worked really hard in this lesson, well done! You're now familiar with the **Array**, a powerful JavaScript object, and some of its fantastic methods. You can also use a **for in loop** to access array values, and you've built a challenging little demo application.

Take your well-deserved break, the next topic awaits you: get ready for an exploration of the very environment that makes JavaScript possible, the **browser**.

Lesson 14: JavaScript Living Space - the Browser Environment

By Maria Antonietta Perna

As we have seen, the JavaScript language is made up of objects, their properties and methods. Your JavaScript code is interpreted and executed by a web browser. Therefore, JavaScript lives and comes alive in the browser environment.

This environment, in its turn, is also present in the JavaScript language, and obviously it is so in object form. Better yet, in the form of several objects, their properties and methods.

In this lesson you will be presented with a quick overview of the main objects making up the browser environment.

In particular, you will learn about:

- the **Browser Object Model (BOM)**;
- the **Document Object Model (DOM)**;
- **accessing, adding, and removing DOM elements and attributes**.

What is the Browser Object Model (BOM)?

The BOM is a a collection of objects that represent the browser and the computer screen. These objects are accessible through the global objects **window** and **window.screen**.

The **window object** is global to the extent that it represents the very host environment of all other JavaScript objects. It's so fundamental, that you've already used it without needing to invoke it explicitly.

In fact, you already used **alert()** and **onload**. You didn't need explicitly to write **window.alert()** because the *global window object* was already presupposed.

Also, you could think about the window object as being at the top of a tree-like hierarchy of other objects, its **descendants, or children**. In fact, it's quite common for programming languages to use family-related metaphors to refer to objects representing their environment.

You will have occasion to work further with the window object through the rest of this tutorial. For now, let's move on to the **Document Object Model (DOM)**.

What is the Document Object Model (DOM)?

One of the most interesting children of the window object, certainly from the point of view of JavaScript, is the **Document object**. This gives access to the all the elements that make up your HTML document.

The DOM is an ever evolving standard defined by the **World Wide Web Consortium (W3C)** implemented with varying degrees of consistency among different kinds of browsers. It contains a wealth of properties and methods that enable you to access and manipulate anything you can think of in an HTML document. So far you've used **document.write() to write on the HTML page, and document.getElementById() to get a hook on an HTML element**. But there's a lot more.

For example, here's all the info the document object can be made to yield when appropriately nudged. Insert the following code inside enclosing <script> tags of an HTML page and preview its content in a browser:

```
//Set up your variables to represent
//bits of info about your document:
var myTitle = document.title;
var myUrl = document.URL;
var myLastModification = document.lastModified;
var myInfoString = "The title of my document is: " + myTitle +
"<br />";
myInfoString += "My document's URL is: " + myUrl + "<br />";
myInfoString += "My document was last modified on: " +
myLastModification;
document.write(myInfoString);
```

That's so cool. Let's have a look at how the HTML page is represented in the DOM.

What does a DOM representation of an HTML page look like?

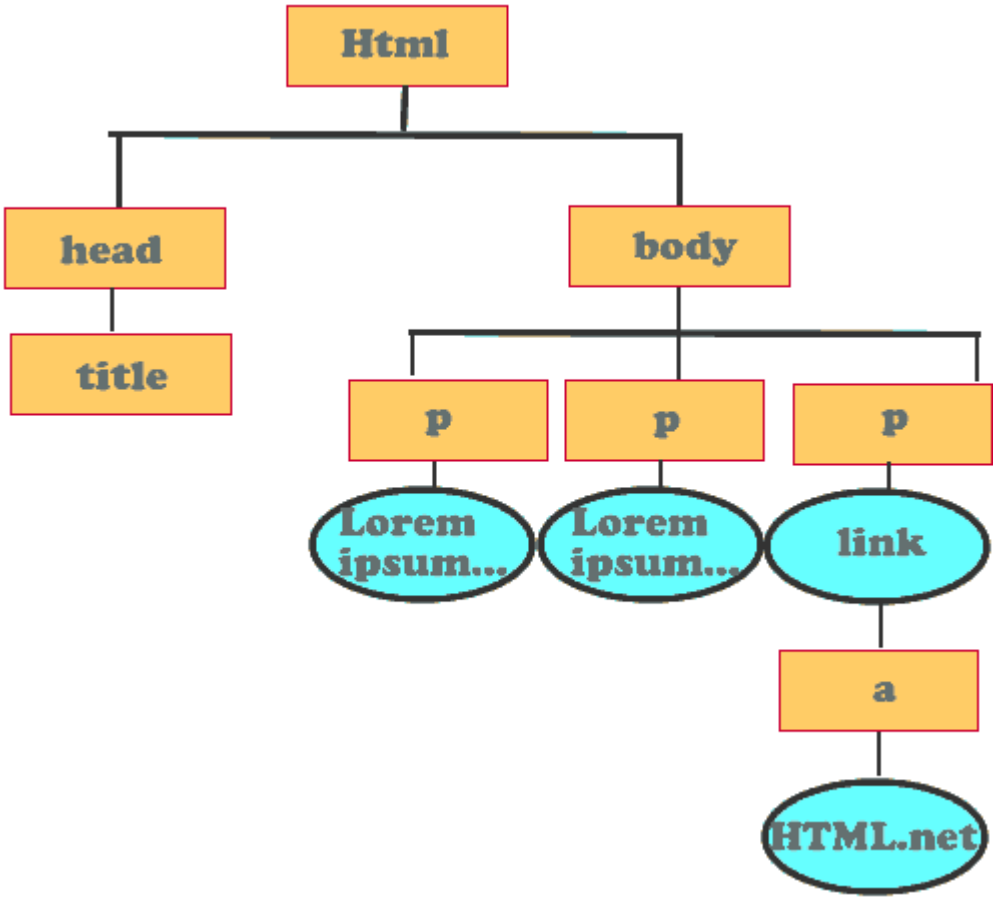
The DOM represents HTML elements in a top-down tree-like structure. For instance, let's consider the following simple HTML document:

```
<html>
<head>
<title>Lesson 14: DOM</title>
</head>
<body>
<p>Lorem ipsum ...</p>
<p>Lorem ipsum ...</p>
```

```
        <p>link <a href="http://html.net">HTML.net</a></p>

        </body>
        </html>
```

The DOM representation of the elements in the HTML page above is as follows:



The DOM elements are also known as **nodes**. Family-related metaphors are also used to represent the relationships and the position of each node with respect to the others. In connection to the DOM tree graphic above, we find:

1. **The <html> node**

   The **<html> element** or **node**, is **parent to all other nodes**. In fact, all other DOM nodes are **nested** inside the <html> node;

2. **<head> and <body> nodes**

   Next, we find 2 elements: the **<head> and the <body>**. These have a **parent/child relationship with the <html> element (the parent)**. Because both are **on the same level, they're siblings** with respect to each other;

3. **<title> and <p> nodes**

   The next level down is occupied by the **<title> as child of the <head>, and 3**

<p> elements as children of the <body>. In our sample HTML page above, <title> and <p> tags are **siblings when considered with respect to each other**;

4. **Text nodes**

   The **text inside the <p> tags, and the text inside the anchor tag** are called **text nodes**. In this case they're children of the <p> element and the <a> element respectively;

5. **Anchor tag**

   The next level down is occupied by an **anchor tag**, which is child to one of the <p> tags.

How do I access DOM elements?

Use one of the 2 following **methods of the document object to access DOM elements**.

Get the DOM element by its id attribute

The most widely used **method of the document object** to access DOM elements is certainly our old friend **document.getElementById(elementId)**. This is the most appropriate way of accessing 1 HTML element at a time.

```
var inputBox = document.getElementById("txtHello");
alert(inputBox);
//if you run this code in an HTML page containing
//an input element called txtHello, you should see
//[object HTMLInputElement] displayed in an alert box.
```

Get DOM elements by tag name

Another popular way of accessing DOM elements is by using their tag name in the HTML mark-up. This is a handy way of grabbing all HTML elements in one go that have a tag name in common, such as images, links, list items, etc. You can store a reference to these elements in an array and manipulate them via JavaScript.

For example, in an HTML page containing a list and 3 list items, you can access each list item and modify their color and background-color style properties by inserting the following code in enclosing <script> tags:


```
//Retrieve all 3 list items and store them in an array
var listElems = document.getElementsByTagName("li");
//loop over each list item and access the style property
for (var i = 0; i < listElems.length; i++)
{
//Notice how you write backgroundColor,
//NOT background-color as you would in CSS:
listElems[i].style.backgroundColor = "red";
listElems[i].style.color = "white";
}
```

Show example

Run your HTML page in the browser. You should see something like the page indicated by following the example link above.

How do I add DOM elements?

In previous lessons you used the practical and easy **innerHTML property** of an HTML element to add new stuff to your web page, be it new mark-up or new text.

However, *innerHTML*, although widely used, has not been recognized as a standard by the W3C. The DOM standard way to add new elements to an HTML page via JavaScript is done as follows. Enter the code below in enclosing <script> tags inside an HTML page:

```
//First create a <p> element with createElement(element):
var newParagraph = document.createElement("p");
//Then add <p> to the <body> tag with appendChild():
document.body.appendChild(newParagraph);
//Now create the text that goes inside the new <p> element
//with createTextNode():
var newTextPara = document.createTextNode("Hello DOM standard");
//Finally, add the new text inside the <p> tag:
newParagraph.appendChild(newTextPara);
```

You can also add attributes to the newly created DOM nodes with **setAttribute(attrType, attrName)**, and later retrieve them with **getAttribute(attrType)**. Using the code snippet above, you can add an id called

newParagraph to the newly created paragraph like so:

```
var newParagraph = document.createElement("p");
//setAttribute has 2 string arguments: the attribute we want to
set
//and the name we want to apply to the attribute.
//Here we create an id attribute called newParagraph:
newParagraph.setAttribute("id", "newParagraph");
document.body.appendChild(newParagraph);
var newTextPara = document.createTextNode("Hello DOM standard");
newParagraph.appendChild(newTextPara);
//getAttribute() has one string argument: the type of attribute
//we want to retrieve:
alert("The Id of the new paragraph is: " +
newParagraph.getAttribute("id"));
```

Show example

Save your work and preview it in a browser. You should see something like the page indicated by following the example link above.

As you can see, there are at least 2 steps you need to take to insert fresh stuff inside an HTML page with JavaScript:

1. **create the new content**: to do this you use either **createElement(element)** in case of an HTML tag, or **createTextNode(text)** in case of new text content;
2. **append the newly created DOM node** to the appropriate location in the HTML document.

I know, it's a bit more involved than the straightforward *innerHTML*, but it's standard-compliant.

How do I remove DOM elements?

That's pretty simple stuff. First grab the DOM node you want to remove, then use **removeChild(elementToRemove)**.

Prepare a simple HTML document with a <p> element and an input button like so:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 14: Delete DOM Element</title>
</head>
<body>
```

```
<h1>Lesson 14: Delete DOM element</h1>

<p id="remove">I'm going to be removed.</p>

<input type="button" value="Delete paragraph"
onclick="deleteParagraph()" />

</body>
</html>
```

Note that the **onclick attribute** of the button element contains a call to the **deleteParagraph()** function. Let's write this function inside enclosing <script> tags in the <body> of our HTML page:

```
function deleteParagraph()

{

//retrieve the element you want to remove:

var remove = document.getElementById("remove");

//use removeChild(elementToRemove) to delete the element

document.body.removeChild(remove);

}
```

Show example

Save your work, run it in a browser, and click the button. You should see something like the page indicated by following the example link above: after you click the button the paragraph should be removed from the page.

Summary

In this lesson you got introduced to the concepts of **Browser Object Model (BOM)** and **Document Object Model (DOM)**. You also learned a few methods of the document object to **access, modify, add, and delete HTML elements in a standard-compliant manner**.

Well done! It's time to take a break and get ready for the next lesson where you tackle your first real-world task: **setting and retrieving cookies**.

Lesson 15: Useful Tasks (I) – Setting and Retrieving Cookies

By Maria Antonietta Perna

You've come a long way in this tutorial. You're ready to look more closely at how JavaScript performs a real world task: **setting and retrieving cookies**.

In this lesson, you will learn:

- **what a cookie is**;
- how you can **store cookies on your website visitors' computer**;
- how you can **retrieve the cookie and use the values stored in it** on the web page.

In the process, you will have occasion to use the **global escape() function**, the **split() method of the String object**, and the **toGMTString() method of the Date object**.

What's a cookie?

The values stored in JavaScript variables are destroyed as soon as the browser is closed or the page is reloaded. Therefore, if you use an input box on your web page to store visitors' names in a variable, that value is lost as soon as they leave the page.

A common feature present in most websites today is some sort of **personalization** mechanism whereby web visitors are recognized. Some instances of this personal touch can be things like greeting returning visitors by their name, indicating how long it's been since the last visit to the website, showing some products that might be of interest to the user based on past purchases, etc.

As you might have guessed, we're dealing with values, and if we want to use them in our JavaScript program, we need a storage place for them. However, all storage places examined so far are temporary. We need something that **persists** in between visits to our web page.

This is where **cookies** enter the scene.

Cookies are strings of text containing small pieces of data about your website visitors. They are stored by the browser in your visitors' computers. Each cookie has a name and an expiration date, and is separated from other cookies by a semicolon (;). The fact that cookies are all stored in the same place, retrieving a specific cookie out of the collection to read its values calls for some little work on our part.

However, before we can retrieve a cookie, we need to write one and store it in the visitor's browser.

How do I set cookies with JavaScript?

Cookie data is stored in the **document object's cookie property**. The data is stored as **"name=value" pairs**, in which the values may not contain any whitespace, commas (,) and semicolons (;). This is why we use the **escape()** method, which enables us to encode the string in **Unicode format** (e.g., a space is represented as %20).

By default the lifespan of cookies is limited to the current browser session. However, by setting an expiry date in the form of an **"expires=date" pair**, we can prolong the cookie's life (unless the user decides to delete the cookies stored on her machine). The date value is a GMT string, which can be obtained by using the **Date object and its toGMTString() method**.

If you insert the code snippet below inside enclosing <script> tags of an HTML page, you set a cookie named *"myCookie"* on your visitor's computer, that **expires after a minute starting from the time of its creation**:

```
//If no cookie is present, we create one
if (! document.cookie)
{
var cookieName = "Test Cookie";
//escape the string to make sure there are
//no unallowed characters
var myCookie = escape(cookieName);
//set the expiry date to 1 minute from creation date:
//get current date and add 1 minute reduced to milliseconds:
var expiryDate = new Date();
expiryDate.setTime(expiryDate.getTime() + (60 * 1000));
//create the myCookie cookie and add it
//to the cookie property of the document object:
//in name=value pairs separated by ;
document.cookie = "myCookie=" + myCookie + ";" + "expires=" +
expiryDate.toGMTString() + ";";
//leave a message to the user:
document.write("Thank you, " + cookieName + " is your cookie");
}
```

That's all set. The browser running a page containing the code snippet above, will store a cookie by the name "myCookie" with a value of "Test Cookie", that expires after 1 minute.

How do I retrieve and use cookies in my web page?

Retrieving cookies to read their values requires the use of a couple of methods. First, you need to **unescape** the name value that you previously escaped. This means that, if your name value contained a space, it was escaped by replacing the space with %20. Now that you're reading the value back from the cookie name, you need to replace %20 with a space character. This is easily done with the **unescape()** method.

Secondly, you need to extract each name-value pair separately from the cookie string. You do this with the **split()** method of the String object.

split() is used to split a string into an array of comma (,) separated substrings. It takes an optional argument called separator. This specifies the character to use for splitting the string.

The code snippet below will check if cookies are present in the cookie property of the document object. If they are, the script extracts and displays the values of the "myCookie" cookie.

```
//continue from the previous code snippet
else
{
var myCookieString = unescape(document.cookie);
//create a comma-separated array of substrings from the string
//containing the cookie values by using
//split() with "=" as separator.
//Remember name/value pairs in the cookie
//string are separated by an = sign:
var dataList = myCookieString.split("=");
//check that the name at index position 0 is
//"myCookie" (remember the cookie name from previous example):
if (dataList[0] === "myCookie")
{
//let's extract the value of myCookie
//by using split() once again.  Now the items in
//dataList array are separated by " , ".
//This will be our separator in the split() method.
```

```
        //Item 1 (index 0) is the cookie name,
        //item 2 (index 1) is the value, which is what we need:
        var data = dataList[1].split(",");
        //Now we have a new array of strings called data.
        //containing only the cookie value ("Test Cookie"),
        //which is at index 0.  We extract it:
        var cookieName = data[0];
        }
        //Now we display a personalized greeting
        document.write("Welcome back, here's your cookie, " + cookieName);
        }
```

Summary

Setting and retrieving cookies to make your website visitors feel welcome is a widely used practice on the web, therefore it's a great new skill to add to your JavaScript toolbox.

Our next useful JavaScript task is how to make sure your website visitors submit a validly filled out form. This is the topic of our next lesson: **form validation**.

Lesson 16: Useful Tasks (II) – Form Validation

By Maria Antonietta Perna

A web form is a staple feature of almost any website today. Nothing can be more frustrating for your visitors than filling out all form fields, clicking the submit button, and have the entire form come back, emptied of all their *carefully* entered values, and displaying a cryptic error message expecting them to start all over again.

Any data that comes from users **must be validated**. This means that programs must check that no empty values are submitted, that submitted values are of the right type and length, etc. However, such checks are necessarily performed *after* the user has filled out and submitted the form to the server.

If the server is busy and takes a while before alerting the user that the form could not be accepted due to errors, users might get quite annoyed and decide not to come back to your otherwise fantastic website.

Thankfully, this rarely happens nowadays, thanks to JavaScript. In fact, although it's crucial that checking user input must be done on the server (e.g., php, .net, ruby, etc.), client-side JavaScript can do a lot to make the experience less painful and time consuming.

JavaScript runs in the user's browser, therefore it executes immediately. There's no need for the form to be submitted to the server before alerting the user that, for example, she has left one of the required fields empty.

In this lesson, you will learn how to:

- make sure users **fill out all required form fields**;
- make sure users **enter a valid email**.

How do I validate empty form fields?

In previous lessons you already performed some validation tasks. For example, you used **isNaN()** to make sure users entered a number data type. You also checked users didn't submit an empty value by using something like **if (inputboxValue == "") { alert user...}**. Let's see what else you could do to prevent users from submitting an empty form field.

Prepare an HTML document with a form and a button, like the one below:

```
<!DOCTYPE html>
```

```
<html>
<head>
<title>Lesson 16: Form validation</title>
</head>
<body>
<h1>Lesson 16: Form validation</h1>

<form id="myForm" action="#" method="post">

<fieldset>

<p><label for="txtName">First Name:</label>

<input type="text" id="txtName" name="txtName" />

</p>

<p><input type="submit" value="Submit" onclick="validate()" /></p>

</fieldset>

</form>

<script type="text/javascript">

</script>

</body>
</html>
```

In between <script> tags write the **validate() function** as follows:

```
function validate()

{

var userName = document.getElementById("txtName").value;

//if the length of the string value is 0,

//it means the user has entered no value:

if (name.length == 0)

{

alert("Please, enter your name");

return false;

}

else

{

alert("Thank you, " + userName);

}

}
```

Save your work and run the page in a browser. Try submitting the form leaving the inputbox empty, and see what happens. Can you post the form? I hope not.

How do I validate email addresses?

Email addresses present a characteristic pattern. We know that they have one **@ character** and at least a **dot (.) character**.

More detailed patterns of email addresses, as well as of domain names, zip codes, telephone numbers, etc., can be validated using a powerful feature called **regular expression**. A treatment of regular expressions is beyond the scope of this tutorial. However, if you're interested the web offers a wealth of material on the subject.

In our code snippet we use some string manipulation methods to make sure the value entered by the user has an *@ character and a dot*.

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 16: Form validation</title>
</head>
<body>
<h1>Lesson 16: Form validation</h1>

<form id="myForm" action="#" method="post">

<fieldset>

<p><label for="txtEmail">Email:</label>

<input type="text" id="txtEmail" name="txtEmail" />

</p>

<p><input type="submit" value="Submit" onclick="validate()" /></p>

</fieldset>

</form>

<script type="text/javascript">

</script>

</body>
</html>
```

Now, change the *validate() function* as follows:

```
function validate()

{

var email = document.getElementById("txtEmail").value;

//still check user entered a value

if (email.length == 0)

{

alert("Please, enter an email address");

return false;
```

```
}
//if the index position of the @  or . characters
//is -1 (any value bigger than this, including 0,
//means the character is present)
else if (email.indexOf("@") == -1 || email.indexOf(".") == -1)
{
alert("Please, enter a valid email address");
return false;
}
//if we get here, all went well
else
{
alert("Thank you");
}
}
```

Save your work and preview the page in a browser. Now, enter an email address with no *@ character* or no *dot charachter*. Is your form submitting the value to the server? Hopefully, it isn't.

Try out: form validation demo

In this try out you will build a JavaScript program that checks user input (name and email) before web form submission. Prepare an HTML page containing a form with 2 inputboxes and a button, and add a paragraph with an id attribute called *result* to display messages to the user, like the one below:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 16: Form validation</title>
<script type="text/javascript" src="lesson16.js"></script>
</head>
<body>
<h1>Lesson 16: Form validation</h1>
<form id="myForm" action="#" method="post">
<fieldset>
<p><label for="txtName">Name:</label>
<input type="text" id="txtName" name="txtName" />
<p><label for="txtEmail">Email:</label>
<input type="text" id="txtEmail" name="txtEmail" />
```

```
</p>
<p><input type="submit" value="Submit" /></p>
</fieldset>
</form>
<p id="result"></p>
</body>
</html>
```

Now create the **lesson16.js** file. This contains 5 functions:

1. **init()** binds the main function, validate(), to the form's onsubmit event;
2. **validate()** examines the results returned by functions validating required fields and email format and displays an appropriate message;
3. **validateRequired(input)** checks the input box has a value. It returns true if it has and false if it hasn't;
4. **validateEmail(email)** checks the email is valid and returns true if it is, false if it isn't;
5. **writeMessage(text)** displays a message to users when something goes wrong. It would be annoying to have alerts popping up every time an error occurs.

```
function init()
{
var myForm = document.getElementById("myForm");
myForm.onsubmit = validate;
}

/*******************************/

//bind init() function to onload event
onload = init;

/*****************************************/

//validate() checks answers from validateRequired()
//and validateEmail() and displays appropriate messages.
//If an error occurs program execution is stopped:
function validate()
{
var name = document.getElementById("txtName").value;
```

```
var email = document.getElementById("txtEmail").value;
//validateRequired() and validateEmail()
//return true/false values: create
//variables to store the result
var isRequiredNameSet = false;
var isRequiredEmailSet = false;
var isEmailValid = false;
//create variable that holds message to display
var message = "";
isRequiredNameSet = validateRequired(name);
isRequiredEmailSet = validateRequired(email);
isEmailValid = validateEmail(email);
//if all values are valid, prepare thank you message
//and allow form submission
if (isRequiredNameSet && isRequiredEmailSet && isEmailValid)
{
message = "Thank you, your details have been successfully
posted!";
}
//if the name field is empty, write message to user and stop
submission:
else if (! isRequiredNameSet)
{
message = "Please, enter a name";
writeMessage(message);
return false;
}
//If the email field is empty, write message to user and stop
submission:
else if (! isRequiredEmailSet)
{
message = "Please, enter an email";
writeMessage(message);
return false;
}
//If the email is not in a valid format (no @ or . characters),
//we write a message to the user and stop program execution:
else if (! isEmailValid)
{
```

```
message = "Please, enter a valid email";
writeMessage(message);
return false;
}
//If we are here validation is successful:
//display the thank you message with an alertbox:
alert(message);
}

/*********************************************/

//This function takes an argument
//that represents the input element and checks
//that it's not empty - it returns true if input is valid
//and false if input is not valid:
function validateRequired(input)
{
var isValid = false;
if (input.length == 0)
{
isValid = false;
}
else
{
isValid = true;
}
return isValid;
}

/*********************************************/

//validateEmail(email) checks the email is in valid
//format and returns true if it is, false if it isn't:
function validateEmail(email)
{
var isValid = false;
if (email.indexOf("@") == -1 || email.indexOf(".") == -1)
```

```
{
isValid = false;
}
else
{
isValid = true;
}
return isValid;
}

/*********************************************/

//writeMessage(text) has the message to
//display to the user as argument, clears any previous
//content from the paragraph with the id of result
//and inserts the appropriate message for display
//using DOM compliant techniques (lesson 14):
function writeMessage(text)
{
var paragraph = document.getElementById("result");
//The content inside the paragraph is the paragraph's
//first child.  We check if there is any content and remove it:
if (paragraph.firstChild)
{
paragraph.removeChild(paragraph.firstChild);
}
//Now we can create and append the new
//message to display to the user:
paragraph.appendChild(document.createTextNode(text));
}
```

Show example

Save your work and preview it in a browser. You should see something like the page indicated by following the example link above.

Try clicking the submit button without filling out the form properly. If all goes as expected, you should not be able to submit your form unless it's values are complete

and appropriate.

Obviously, your script can't verify an email address, but it can validate its format. Finally, keep in mind that for a detailed check of the email pattern, the best tool for the job is a **regular expression**.

Summary

You've come a long way in your JavaScript journey, well done!

Now get ready for the next lesson: time to start exploring the world of JavaScript animation with the **timing functions**.

Lesson 17: JavaScript Timing Events

By Maria Antonietta Perna

The **global window object** offers a few little functions that are great if you want to create the effect of movement or of passing time on the web.

In this lesson, you will learn about:

- **setTimeout()**;
- **clearTimeout()**;
- **setInterval()**;
- **clearInterval()**.

In the process, you will also learn how to **preload images** with JavaScript and how to **build a simple photo gallery application**.

How do I use setTimeout()?

If your program has a chunk of code that needs to be executed after a specified time, then **setTimeout(actionToExecute, timeInMilliseconds)** is your function.

Let's see it in action. In this simple example, we will call up an alertbox 2 seconds after page load. Type the code below in enclosing <script> tags on a simple HTML page:

```
//Package the code to insert in the
//timer in a simple function:
function sayHello()
{
alert("Hello");
}
//Package the timer into its own function
function timeGreeting()
{
//assign the timer to a variable:
//you need to do this so the timer can be
//referenced by another function when you want
//to stop it:
var greeting = setTimeout(sayHello, 2000);
//Notice how the time is set in milliseconds.
}
```

```
//Now call the timer function
timeGreeting();
```

Save your work and run it in a browser. After 2 seconds, an alertbox should pop up to greet you.

How do I use setInterval()?

If you need to execute a chunk of code at specified time intervals, then **setInterval(codeToExecute, timeIntervalInMilliseconds)** is your function.

To see it in action in its most annoying manifestation, just replace **setTimeout** with **setInterval** in the previous example and you're done.

Now you should see an alertbox greeting you every 2 seconds! That's enough, just close that window in your browser and let's modify our script so that we can put a stop to that nuisance.

How do I stop the timer?

JavaScript offers 2 handy methods to get rid of timers: **clearTimeout(timerVariable)** if your timer uses setTimeOut(), and **clearInterval(timerVariable)** if your timer uses setInterval(). Let's put *clearInterval()* to good use right away by stopping that annoying alertbox. First of all, add an inputbox to your HTML page, like so:

```
<input type="button" value="Stop timer" onclick="stopTimer()" />
```

Next, rewrite the preceding JavaScript code and then add the new function as follows:

```
//Make the variable holding the timer
//global, so that it can be accessed both
//by the function setting the timer and by the function
//stopping the timer:
var greeting;

//sayHello() remains unchanged from the previous example
function sayHello()
{
alert("Hello");
}

//we increase the time interval
```

```
//to give more time to the user to
//click the button that stops the alert:
function timeGreeting()
{
greeting = setInterval(sayHello, 5000);
}

timeGreeting();

//package the code that cancels the timer
//in its own function that will be called when
//the button on the page is clicked:
function stopTimer()
{
//you call clearInterval() and pass the variable
//containing the timer as argument
clearInterval(greeting);
}
```

Save your work and preview it in a browser. Just click the button and the annyoing alertbox disappears: that's much better, great! I leave you to experiment with clearTimeout() on your own.

Try out: a photo gallery application

JavaScript timers are often used to produce animation effects on the web. Photo galleries are one of those widely used applications that often employ JavaScript timers.

It's easy to find a number of sophisticated photo gallery widgets on the web, that you can download and readily plug into your own web page.

However, it's still nice to be able to build a simple prototype, just to really challenge yourself with your newly gained knowledge. In any case, your application can be considered a work in progress, something that evolves and grows as your JavaScript programming skills evolve and grow.

At this stage, your photo gallery will consist of a photo container and 2 buttons: one button to stop the gallery, and one button to restart the gallery. As soon as the page loads, the gallery automatically displays its images one at a time every 2 seconds.

You will also need a few images, possibly all of the same size. For this demo, I prepared four 620px X 378px graphics and saved them in their own *images* folder.

Let's start from the HTML page. This is made of a , an <img> tag, and 2 <input> tags. Also included are: **1) a small style declaration in the <head> of the document** to the effect that the element that contains the gallery will be centered and sized to the same width and height as the gallery graphics; and **2) a reference to an external JavaScript file at the bottom of the <body> element of the HTML page**. This location is the most appropriate one because we need to make sure the HTML page and, most importantly, the image referenced by the <img> tag, are fully loaded before the script kicks in.

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 17: JavaScript Timing Events</title>
<style type="text/css">
  #gallery
        {
        width:620px;
        height:378px;
        margin: 0 auto;
        border:2px solid #ccc;
        }
</style>
</head>
<body>
<h1>Lesson 17: My Photo Gallery</h1>

<div id="gallery">

<img src="images/1.jpg" alt="Photo gallery image" id="photo" />

<input type="button" id="btnStart" value="Restart gallery" />

<input type="button" id="btnStop" value="Stop gallery" />

</div>

<script type="text/javascript" src="lesson17.js"></script>

</body>
</html>
```

Now prepare the **lesson17.js** file. Your JavaScript code consists of 3 functions:

1. **init()** contains initialization routines: it preloads the photo gallery images so that they will be ready for display as soon as the script calls them. In addition, it binds event handlers to the 2 button elements to stop and restart the gallery;
2. **startGallery()** displays each graphic in the gallery every 2 seconds;
3. **stopGallery()** stops the gallery so that the photo that comes next with respect to the current photo is not displayed.

Furthermore, the code contains a few global variables at the top that need to be accessed by all the functions in the script. Let's get started.

```
//Global variables: a reference to the

//photo currently displayed on the page,
```

```
var curImage = document.getElementById("photo");
//a variable to store the timer,
var galleryStarter;
//a variable to store a true/false value indicating
//to the program whether the gallery is on or off,
var isGalleryOn = true;
//an array containing 4 strings representing the filepaths
//to the image files in the images folder,
var images = ["images/1.jpg", "images/2.jpg", "images/3.jpg",
"images/4.jpg"];
//an empty array that will be filled with 4 preloaded
//image objects: the src property of these image objects will
correspond
//to the filepaths contained in the images[] array,
var preloadedImgs = [];
//a variable that works as our counter to
//advance from one image to the next.  It starts at 0.
var counter = 0;

/*************************************/

//Init() starts with the image preloading routine.
//First fill the preloadedImgs[] array with a
//number of image objects corresponding to the length
//of the images[] array:
function init()
{
for (var i = 0; i < images.length; i++)
{
preloadedImgs[i] = new Image();
}
//now assign the value of the strings contained in the
//images[] array to the src property of each image object
//in the preloadedImgs[] array, using one more loop:
for (var i = 0; i < preloadedImgs.length; i++)
{
preloadedImgs[i].src = images[i];
}
```

```
//Now, assign event handlers to the 2 buttons
//to restart and stop the gallery:
var btnStart = document.getElementById("btnStart");
var btnStop = document.getElementById("btnStop");
btnStart.onclick = startGallery;
btnStop.onclick = stopGallery;
//Finally, check the isGalleryOn flag is true.  If it is
//call the function that starts the gallery:
if (isGalleryOn)
{
startGallery();
}
}

/*********************************************/

//Assign the init() function to the onload event
onload = init;

/*********************************************/

//startGallery() contains the functionality
//to extract the photos from the preloadedImgs[] array
//for display and to set the timer in motion:
function startGallery()
{
//extract the image filepath using the counter
//variable as array index and assign it to the src
//property of the curImage variable:
curImage.src = preloadedImgs[counter].src;
//advance the counter by 1:
counter ++;
//if the counter has reached the length of the
//preloadedImgs[] array, take it back to 0, so the
//photo gallery redisplays the images from the start:
if (counter == preloadedImgs.length)
{
```

```
counter = 0;
}
//Set the timer using this same function as one
//of the arguments and 2000 (2 milliseconds) as the other
argument.
galleryStarter = setTimeout("startGallery()", 2000);
//Signal that the gallery is on to the rest of the program:
isGalleryOn = true;
}

/************************************************/

//stopGallery() uses clearTimeout()
//to stop the gallery
function stopGallery()
{
clearTimeout(galleryStarter);
//signal that the gallery has stopped to the
//rest of the program:
isGalleryOn = false;
}
```

Show example

Save your work and preview it in a browser. You should see something like the page indicated by following the example link above. Try stopping and restarting your gallery.

Questions, questions, questions

I imagine you might be wondering what this business of **preloading images** is all about. Well, here's a short explanation.

When an HTML page is first loaded, all the resources needed by the page, including external CSS files, scripts, images, videos, etc., are downloaded as well.

In the case of inserting a resource, such as an image, dynamically into an HTML page after the page has fully loaded in the browser, as it's the case with our photo gallery, the following issue might arise: the requested image has to be sent to the client browser by the server, and this might take a few seconds. Although this is not terrible, it might seriously spoil the *magic JavaScript effect* we intend to create.

Here's where **preloading** the images in JavaScript comes to the rescue. By doing the preloading routine, the image object is created on page load, and by assigning the image filepath as value to the image object's src property, we make sure that the image we intend to add dynamically to the page is ready to be grabbed by our script for immediate use.

This is the image preloading routine simplified to the bare minimum:

```
//Create an Image object:
var myNewImage = new Image();
//assign the filepath of your desired image
//to the src property of the newly created image object:
myNewImage.src = "images/testimage.jpg";
//Now myNewImage has been preloaded: your
//JavaScript can use it, display it, etc. without fear of
//having to wait too long for a server request.
```

Summary

You've made it to the end of the lesson, congratulations! You've learned a great deal: you can set and stop timers with JavaScript, preload images for dynamic display, and build an automatic photo gallery with JavaScript.

It's time for your break before tackling your next lesson's topic, a major feature of contemporary web applications: **AJAX**.

Lesson 18: Making AJAX Calls

By Maria Antonietta Perna

One core feature of 21st century web applications is their fantastic responsiveness. A pioneer in the business of making the web more and more similar to a desktop application has been *Google*: as soon as you start typing a character in a Google box, all kinds of suggestions and search results keep popping up to speed up your search; and again, just type something using Google maps, and you're instantly catapulted to almost any street on the planet.

Behind this amazing feat of human cleverness is the coming together of a few technologies collectively known as **AJAX**.

In this lesson you will learn:

- what **AJAX stands for** and **what it is all about**;
- how to **make an AJAX request**;
- how to **handle an AJAX response**.

I recommend you get access to a web server to carry out the example code illustrated in this lesson. In fact, AJAX is all about client-server communication. In view of this, you can upload your files to a hosted server or to a local server. The PHP tutorial on this website provides an excellent guide on this topic.

What's AJAX all about?

The acronym AJAX stands for **Asynchronous JavaScript And XML**.

In actual fact, it's a combination of internet standards made up of:

- **standards-based presentation** using **HTML and CSS**;
- **dynamic display** using the **DOM**;
- **data interchange** and manipulation using **XML**;
- **asynchronous** data retrieval using the **XMLHttpRequest object**;
- **JavaScript** magic to orchestrate the whole process.

In non-AJAX web applications, the interaction between servers and clients can be a tedious business:

1. a user action from the client sends a request to the web server via HyperText Transfer Protocol (HTTP);
2. the web server receives the request and processes it by invoking server-side scripts, database data, etc., and sends a response back to the client via HTTP;
3. the client browser receives the response and loads the entire updated page.

Having to go from browser to server and back again each time a user requests some piece of data from the server, in addition to undergoing an entire page refresh at each update, can be quite stressful on servers and on users alike.

AJAX helps in at least 2 respects: **piecemeal page updates** and **asynchronous communication** between server and client.

What this means in a few words, is that every time the user interacts with the web page, only those bits that need updating are refreshed, not the entire page. Furthermore, the fact that AJAX operations are performed asynchronously, means that during the time that it takes for the server to respond, the page is not locked and the user can still interact with the website.

How do I make an AJAX request?

An **AJAX request** is made using the **XMLHttpRequest object** and its **open() and send() methods**. This is supported by all major browsers. However, older browsers, namely older varsions of Microsoft Internet Explorer (vesions 5 and 6), support an **ActiveXObject**. This little hurdle is easily overcome by testing for feature support in the script.

The **open(retrievalMethod, url, bool)** method has 3 arguments:

1. **retrievalMethod**: this can either be a **GET** (used to fetch data from the server), or a **POST** (used to send data to the server);
2. **url**: this is the location where the data is made available. It can be a text file, an XML document, or a server-side script that processes data coming from a database;
3. **bool**: this is a true/false value. If it's false the request is made synchronously, if it's true the request is made asynchronously, which is what we usually want.

The XMLHttpRequest object has an **onreadystatechange** property that deals with the response from the server. This proceeds over the following 5 stages:

- 0) the **request is uninitialized** because **open() has not been called**;
- 1) the **request is specified**, but **send() has not been called yet**;
- 2) the **request is being sent**, because now **send() has been called**;
- 3) the **response is being received, but not yet completed**;
- 4) the **response is complete and data is available for manipulation and display**.

Upon completion (stage 4), the XMLHttpRequest object's **status property** gets assigned an **HTTP status code** that describes the result of the request as follows:

- **200**: success!
- **401**: unauthorized - authentication is required and was not provided;

- **403**: forbidden - the server refuses to respond;
- **404**: not found - the requested resource cannot be found.

This is a simple code snippet that translates what's just been said into practice:

```
//Global variable to store the XMLHttpRequest object
var myRequest;
//Package the request into its own function
function getData()
{
//Feature-testing technique to make sure the browser
//supports the XMLHttpRequest object
if (window.XMLHttpRequest)
{
//create a new instance of the object
myRequest = new XMLHttpRequest();
}
//else - the XMLHttpRequest object is not supported:
//create an instance of ActiveXObject
else
{
myRequest = new ActiveXObject("Microsoft.XMLHTTP");
}
//Use the open(retrievalMethod, "myDataFile.txt", bool) method
myRequest.open("GET", url, true);
//Use send() with a null argument - we have nothing to send:
myRequest.send(null);
//attach a function to handle onreadystatechange,
//that is, the response from the server:
myRequest.onreadystatechange = getData;
}
```

How do I handle the AJAX response?

If the client browser requests text data, the server response is automatically stored in the **responseText property**.

If the client browser requests data in XML format, the server response is stored in the

responseXML property.

Here's a code snippet that illustrates this in practice.

```
//Package the response-handling code in a function
function getData()
{
//Make sure the response is at the completion stage (4):
if (myRequest.readyState ===4)
{
//if it is, make sure the status code is 200 (success):
if (myRequest.status === 200)
{
//if all is well, handle the response:
var text = myRequest.responseText;
alert(text);
}
}
}
```

Let's tackle two examples that include both request and response using AJAX. The first example will be dealing with a response in plain text format, the second example with a response in XML format.

AJAX request - response example: plain text response

To follow along, you need a server and 1 simple text file uploaded to the same server that hosts your JavaScript code.

The page you're going to build allows the user to click a button to display some text stored in your text file using AJAX.

Here's a simple HTML page with a <header> tag and a button element:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 18: Making AJAX Calls</title>

</head>
<body>
<h1>Lesson 18: Making AJAX Calls - Plain Text Response</h1>
```

```
<div>
<h2 id="myHeader">Click the button to call your data</h2>
<input type="button" value="Click Me!"
onclick="getText('lesson18_test.txt')" />
</div>
<script type="text/javascript">
//JavaScript AJAX code here
</script>
</body>
</html>
```

Here's the JavaScript code that goes in enclosing <script> tags in the HTML page above:

```
//Prepare the global variable for the request
var myRequest;
//Write the getText(url) function
function getText(url)
{
//check support for the XMLHttpRequest object
if (window.XMLHttpRequest)
{
myRequest = new XMLHttpRequest();
}
//else, create an ActiveXObject for IE6
else
{
myRequest = new ActiveXObject("Microsoft.XMLHTTP");
}
//Call the open() method to make the request
myRequest.open("GET", url, true);
//Send the request
myRequest.send(null);
//Assign the getData() function to the
//onreadystatechange property to handle server response
myRequest.onreadystatechange = getData;
}
```

```
/*************************************************/

//This function handles the server response
function getData()
{
//Get a reference to the header element where
//the returned result will be displayed
var myHeader = document.getElementById("myHeader");
//Check the response is complete
if (myRequest.readyState ===4)
{
//Check the status code of the response is successful
if (myRequest.status === 200)
{
//Store the response
var text = myRequest.responseText;
//Assing the returned text to the nodeValue
//property of the header element (you can also use
//innerHTML here if you feel it simplifies your task)
myHeader.firstChild.nodeValue = text;
}
}
}
```

Show example

Save your work and preview it in a browser. You should see something like the page indicated by following the example link above.

Notice how the result is returned from the server with no page flashing, no change of URL, no whole page refresh: that's the AJAX magic.

AJAX request - response example: XML format response

The page you're going to build in this second example allows the user to click a button to display a series of programming languages coming from an XML file using AJAX.

To follow along in this exercise, prepare a simple XML document and an HTML page.

Here's what my XML document looks like:

```
<?xml version="1.0" ?>

<languages>
 <language>PHP</language>
 <language>Ruby On Rails</language>
 <language>C#</language>
 <language>JavaScript</language>
</languages>
```

Save the file above as **lesson18_test.xml** in the same location as your HTML page.

Here's the HTML page:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 18: Making AJAX Calls</title>

</head>
<body>
<h1>Lesson 18: Making AJAX Calls - XML Response</h1>

<div id="test">

<h2>Click the button to get a list of programming languages</h2>

<input type="button" value="Click Me!"
onclick="loadXml('lesson18_test.xml')" />

</div>

<script type="text/javascript">

//JavaScript AJAX code here

</script>

</body>
</html>
```

Now proceed with the JavaScript code as follows:

```
var myRequest;

//The greatest part of the loadXML() function

//is similar to the previous example:

function loadXML(url)

{

if (window.XMLHttpRequest)

{

myRequest = new XMLHttpRequest();

}

else
```

```
        {
        myRequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
        myRequest.open("GET", url, true);
        myRequest.send(null);
        myRequest.onreadystatechange = getData;
        }

        /*****************************************/

        function getData()
        {
        //Get a reference to the div element
        //where the returned data will be displayed:
        var myDiv = document.getElementById("test");
        //The part of the code that checks the response
        //is the same as the previous example:
        if (myRequest.readyState ===4)
        {
        if (myRequest.status === 200)
        {
        //Use the responseXML property to catch
        //the returned data, select the xml tag
        //called language, and store returned
        //language items in an array variable:
        var languages =
myRequest.responseXML.getElementsByTagName("language");
        //Loop over each array item containing the data
        //and create a <p> element to contain each returned
        //piece of data.  Notice the use of firstChild.data
        //to retrieve the value of the XML <language> tag:
        for (var i = 0; i < languages.length; i++)
        {
        var paragraph = document.createElement("p");
        myDiv.appendChild(paragraph);

paragraph.appendChild(document.createTextNode(languages[i].firstChild.data));
        //You can also assign the returned result to myDiv.innerHTML
        //if you feel it would be simpler.
```

```
        }
    }
}
}
```

Show example

Save your work and preview it in a browser. You should see something like the page indicated by following the example link above.

Notice how the result is returned from the server without a whole page refresh. That's terrific!

Summary

That's it for this lesson. You worked really hard, but now you can write JavaScript code that retrieves data from the server without a whole page refresh using AJAX techniques.

You're not limited to text format files or to XML files, though. In fact, databases are the most commonly used tools to permanently store data on a server. AJAX techniques are as easily applied to deal with database retrieved information. However, you'll need PHP or a server-side programming language to retrieve data from or post data to a database.

In any case, we will return to this topic in our last lesson and see how jQuery greatly simplifies the whole *Ajaxification process*.

But first, we need to get some familiarity with the most popular library in the JavaScript programming world today, **jQuery**, which is the topic of our next lesson.

Lesson 19: Short Introduction to jQuery

By Maria Antonietta Perna

You've come a long way in your JavaScript journey. Now it's time to get to the fun part.

This lesson, together with the next 2 lessons, will introduce you to **jQuery**, the fantastic JavaScript library invented by **John Resig**.

However, this is a wide topic and these 3 lessons cannot but give you just a taste for what jQuery can really do.

Therefore, I encourage you to visit the jQuery website after you finish this tutorial, and explore and experiment with jQuery's numerous methods, events, effects, etc., for yourself.

In this lesson you will learn:

- about the **benefits of using jQuery**;
- how to **include jQuery in your project**;
- about **jQuery selectors** to manipulate DOM elements in code;
- how **jQuery handles events**;
- how to use **anonymous functions** with jQuery;
- how to **add** and **remove DOM elements** with jQuery.

Why should I use jQuery?

Here are some good reasons why using jQuery in your JavaScript projects is a great idea.

jQuery is:

- extremely light-weight (just 31 kb in its latest minified and zipped production-ready version);
- it's free and very easy to include in your projects: just download its latest version from the jQuery website, or use an online Content Delivery Network;
- it's continually upgraded, maintained and documented by a dedicated community of great developers. This ensures high quality and support on the internet;
- it helps overcoming inconsistencies in the way some JavaScript features are implemented across different browsers;
- and last but not least, it offers a wealth of ready-made animation effects that are a joy to use.

How do I include jQuery in my website?

Including jQuery in your project is fast and easy. You have the choice of downloading a copy of the library and save it locally, or you can use a **Content Delivery Network (CDN)** to serve jQuery to your website on the Internet.

Go to the jQuery website download page

How to include a local copy of jQuery

If you prefer having a local copy of jQuery:

1. go to the *Download jQuery* section of the download page and click on the **minified version of the latest release** of jQuery. At the time of writing this is **version 1.7**;
2. copy and paste the code in a text file;
3. save the file in your web project directory (better if you save it in its own subfolder);
4. in your html page include the appropriate <script> tags in the following order:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 19: Introduction to jQuery</title>

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

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

</head>
<body>
<h1>Lesson 19: Introduction to jQuery</h1>

</body>
</html>
```

The reference to jQuery comes before your own JavaScript code file. In fact, your code needs to find jQuery already fully loaded to be able to use it.

How to include a hosted copy of jQuery

If you decide to opt for a CDN to serve up a copy of jQuery to your website, then follow these easy steps:

1. go to the *CDN Hosted jQuery* section of the download page and pick one of the different CDN services available;
2. copy the URL of your CDN of choice and paste it as the value of the **<script> tag's src attribute**, like so:

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

3. place a <script> tag with a reference to your own JavaScript file underneath the jQuery <script> tag, like you did in the previous example.

That's it, you're done!

How to make sure jQuery is working

Now the jQuery library can be used in your project. To make sure things work as expected, create a text file and call it *jquery_test.js*. Enter the following code:

```
//Most of your code goes inside
//the jQuery ready() function:
$(document).ready(function()
{
alert("jQuery is working!");
});
//Make sure you keep track of braces, brackets, and semi-colons ( ; )
```

Save all your files and run the HTML page in a browser. If you typed the code exactly like the example above, you should be greeted by ... well, the good old alert box.

The **$ sign** is an alias for jQuery. You could replace it with the keyword *jQuery* and your jQuery-powered JavaScript code would work just fine. However, less typing is jQuery's (and all coders') motto, so using *$* is the accepted convention.

```
.ready(function() { //JavaScript code here ... } );
```

is where most of your JavaScript code will be placed. This is jQuery's clever way of making sure your code runs when the document is ready for it: that is, when the html is fully loaded.

How do I access DOM elements using jQuery?

jQuery appeals so much to web designers also because it enables us to select DOM elements in code just like we do with CSS. Let's see it in action.

Select html elements by id

Remember our old document.getElementById()? Well, jQuery makes selection by id much shorter. Here's how it's done:

```
$(document).ready(function()
{
var myDiv = $("#myDiv");
});
```

The *$ sign* in front of *("#myDiv")* can be translated like: "Hey, browser, get me the element with the id of myDiv!"

As you can see, the id attribute is grabbed by using the # sign, the same as CSS.

Select html elements by tag name

You also came across document.getElementsByTagName(). This method allows us to grab all html elements by their tag.

Look how jQuery makes it a lot snappier:

```
$(document).ready(function)
{
//store all images on the page in an array
var images = $("img");
});
```

As you can see, the magic $ does most of the talking.

Select html elements by class name

A quick way to target html elements that share the same class is as follows:

```
$(document).ready(function()
{
var redElements = $(".classRed");
});
```

Now *redElements* contains an array of all the html elements on your page with a class of *classRed*.

Once again, notice how jQuery facilitates selection by using the same notation you use in your **CSS declarations to target classes: .className**.

Select html elements using jQuery filters

jQuery enables you to be as precise as a surgeon when targeting html elements and attributes on the page. jQuery offers a **filtering syntax to query the DOM** that is both simple and efficient.

Here's a list of the most common filters:

Filter	Description
:first	The first item in a matched set `$('p:first')` returns the first paragraph.
:last	The last item in a matched set `$('p:last')` returns the last paragraph.
:odd	The odd-indexed items in a matched set `$('tr:odd')` returns table rows indexed at 1, 3, etc.
:even	The even-indexed items in a matched set `$('tr:even')` returns table rows indexed at 0, 2, etc.
:has	finds elements having the child element specified `$('p:has(span)')` returns all paragraphs containing a span element.
:eq	returns the element with the matched index `$('p:eq(1)')` returns the second paragraph starting to count at 0.
:contains(x)	Returns all elements containing the text x `$('div:contains(foo)')` targets all divs with the text 'foo'.

A complete list of all selection methods and filters is available on the jQuery website. Have a look and experiment with the available code samples to gain more familiarity with the new approach.

How do I assign event handlers using jQuery?

Event handlers are code blocks, usually functions, that specify what your application should do when a specified event like, for example, a button click, occurs.

So far, you've used 2 approaches:

1. **hard-wiring a function to the html element itself**: `<input type="button" onclick="doSomething()" />`
2. **assigning a function to the DOM object's appropriate property**: `myElement.onclick = doSomething;`

Both approaches are fine. However, the first one sins against the **separation of concerns principle**. In fact, it mixes in JavaScript code with HTML code.

The second approach complies with the separation of concerns principles, but comes short in case we want to attach more than a function to the same element.

There's actually a third approach that overcomes all shortcomings but one: it's implemented differently in Internet Explorer browsers with respect to all other major browsers.

The good news is: jQuery is here to make event handling quick and easy. Here are a few jQuery approaches to choose from.

The bind() method

bind(eventType, handler) works as follows.

Suppose you have an html element with an id of *myElement*, and a function called *sayHello()*. You want sayHello() to kick in when the user clicks on the html element. This is how you achieve this using bind():

```
$(document).ready(function()
{
//create the sayHello function
function sayHello()
{
alert("Hello jQuery");
}
//Attach the handler using .bind():
myElement.bind('click', sayHello);
});
```

You can find more details on the bind() method on http://api.jquery.com/bind/.

The ready-made approach

jQuery offers some ready-made handlers corresponding to JavaScript events. Here's a list of the most widely used ones:

Handler	Description
click()	Binds a handler to an **onclick** event.
hover()	Binds a handler to the **onmouseover** and **onmouseout** events.
change()	Binds a handler to the **onchange** event (when the content of a field changes).
select()	Binds a handler to the **onselect** event (when text is selected).
submit()	Binds a handler to the **onsubmit** event of a form element.
focus()	Binds a handler to the **onfocus** event (when an element gets focus).
keypress()	Binds a handler to the **onkeypress** event (when a key on the computer keyboard is pressed).

Assuming you have an html element with an id attribute of *myElement*, and a function called *sayHello()* - as in the previous example - you can bind sayHello() to the onclick event of myElement inside jQuery's ready() function like so:

```
myElement.click(sayHello);
```

The on() method

With this latest version of jQuery (v.1.7), a brand new method to handle events has been introduced.

The **on()** method is highly recommended by the jQuery team if you start a new project.

It can be used to attach handlers both to elements already present in the HTML page from the start and on dynamically added elements (unlike bind()). Here's how it's used.

Using the same *sayHello() function*, inside jQuery's .ready() function simply write:

```
myElement.on('click', sayHello);
```

More details on the **on() method** can be found on http://api.jquery.com/on/.

Anonymous functions

Code samples using jQuery often employ **anonymous functions**. These are functions without a name, and can be quite handy, although I find names more helpful in terms of code readability. If we replace the *sayHello()* function with an anonymous function, we have:

```
//Using bind():
myElement.bind('click', function()
{
alert("Hello jQuery");
});

/*********************************/

//Using click(), we have:
myElement.click(function()
{
alert("Hello jQuery");
});

/*************************************/

//Using on(), we have:
myElement.on('click', function()
{
```

```
alert("Hello jQuery");
});
```

Add and remove DOM elements

Once grabbed by your JavaScript code, DOM elements can easily be manipulated with jQuery.

You've had a taste of standards-compliant DOM manipulation techniques in this tutorial. Therefore, you will readily appreciate the jQuery way of performing the same tasks.

Add new DOM elements

You can both retrieve html elements and add new html elements to the page using jQuery's **.html()** method. When you don't pass any arguments, html() retrieves a DOM element, if you pass a string argument, html() adds that string to the page.

If you just need to add new text inside an html element, you can use jQuery's **.text()** method and pass a string argument representing the text you intend to add. Like .html(), you can use .text() without passing any arguments to retrieve text content from the HTML page.

To test .html() prepare an HTML page containing a tag with an id of *myElement*, a <p> tag and some text. First we retrieve the existing paragraph element, then we create a new paragraph.

```
$(document).ready(function()
{
//Use html() to retrieve content:
var pageParagraph = $('p').html();
//the alert should display <p>'s content
alert(pageParagraph);
});

/************************************************/

//Now let's change the div's content
//by adding a new paragraph.
var newParagraph = $('#myElement').html('<p>Hello new
Paragraph!</p>');
//close off all your brackets:
});
```

Save all your files and preview the result in a browser. The alert should display the original paragraph's content, and the page should subsequently display the new paragraph's content.

The way you implement .text() is the same as .html(). I leave you to experiment with it on your own. More details on .html() can be found on http://api.jquery.com/html/, and on .text() can be found on http://api.jquery.com/text/.

Remove DOM elements

A simple way of doing this is with jQuery's **remove()** method. Let's put it to the test.

Use the same HTML page from the previous example and add a button element with an id of *btnDelete*.

In a fresh JavaScript file, add the following code:

```
$(document).ready(function()
{
//use a click handler and an anonymous function
//to do the job
$('#btnDelete').click(function()
{
//delete all <p> elements
$('p').remove();
});
});
```

Save all your files and preview the HTML page in a browser. Click the button and see the paragraphs on the page disappear.

More on **.remove()** can be found on http://api.jquery.com/remove/.

Summary

There's a lot to digest in this introduction to jQuery and you've done a great job coming as far as you did.

jQuery is such a rich library that it can't be covered in a few lessons. Therefore, I invite you to practice with all the code samples in this lesson and the following lessons as much as you can.

Also, it's crucial that you explore the jQuery website and experiment with the code samples available in their documentation and tutorials. If you have doubts, don't hesitate to send us a line in the forum dedicated to this tutorial.

More on jQuery coming up in the next lesson: get ready for some fabulous **JQuery effects** in just a few lines of code!

Lesson 20: Cool Animation Effects with jQuery

By Maria Antonietta Perna

jQuery is so rich with ready-made functions that adding fancy effects on your web page is a breeze. It's hard not to get carried away with it.

Here, we are going to explore the most commonly used jQuery effects. Once again, my recommendation is that of integrating this lesson with a visit to the jQuery website for more code samples and detailed explanations.

In this lesson you will learn how to:

- add **effects by dynamically manipulating styles**;
- use **jQuery's show()/hide()/toggle()** methods;
- use **jQuery's fadeIn()/fadeOut()/fadeToggle()** methods;
- use **jQuery's slideUp()/slideDown()/slideToggle()** methods.

In the process, you will build a simple **sliding menu** and get more practice with the new jQuery approach.

How do I add and remove CSS classes with jQuery?

jQuery enables you to manipulate CSS styles in code with **$.css(styleName, styleValue)**. It also enables you to add and remove CSS classes in relation to DOM elements.

Because $.css() would sprinkle your HTML page with inline CSS style rules, we will focus on applying and removing CSS classes with **$.addClass(className)**, **$.removeClass(className)**, and **$.toggleClass(className)**. This approach uses the CSS class declarations stored either in the <head> of your HTML page or in a separate file, therefore it's better suited to current best practices in web design.

If you're curious about $.css(), by all means visit http://api.jquery.com/css/ for more details.

Suppose you want to replace the CSS class *bigDiv* with the CSS class *smallDiv* when the user clicks inside the div element. This is how you could do the task with jQuery:

```
$(document).ready(function()
{
//Store a reference to the div in a variable using its id:
```

```
var myDiv = $('#myDiv');
//Attach a click handler to manipulate the CSS classes:
myDiv.click(function()
{
//Remove the existing CSS class and replace it with the new one.
//Because the element is the same, chain both methods:
myDiv.removeClass('bigDiv').addClass('smallDiv');
});
});
```

What about toggling between the CSS bigDiv and smallDiv classes by clicking inside the div? That's easy: jQuery has this little handy method for you - **$.toggleClass()**. Assuming the CSS class *bigDiv* is hard-coded in the class attribute of the div element, you can toggle the CSS class *smallDiv* using jQuery, like so:

```
$(document).ready(function()
{
var myDiv = $('#myDiv');
myDiv.click(function()
{
myDiv.toggleClass('smallDiv');
});
});
```

Now, keep clicking inside the div element, and you'll see it getting smaller and bigger at each click of the mouse.

How do I use jQuery show()/hide()?

If you want to show/hide HTML elements on a page, you can do so easily with jQuery's **show()/hide()**. You can also modulate the way elements are revealed or hidden by adding the **'slow'** or **'fast'** arguments, or even the **number of milliseconds** you'd like the transition to last.

Let's say you have a paragraph element inside a div with an id of *myDiv*. You want the paragraph to slowly disappear when the user clicks inside the div element. Here's how you could accomplish this with jQuery **hide()**.

```
$(document).ready(function()
```

```
{
var myDiv = $('#myDiv');
myDiv.click(function()
{
//Replace 'slow' with 2000 for a slower transition.
$('p').hide('slow');
});
});
```

Clicking inside the div element results in the paragraph slowly disappearing from view. jQuery **show()** works the same way. I leave you to experiment with it on your own.

What if you wanted to toggle between show/hide transitions as the user clicks inside the div element? No problem, jQuery offers the **$.toggle()** method. To see it in action, simply replace *hide()* with *toggle()* in the previous example, like so.

```
$(document).ready(function()
{
var myDiv = $('#myDiv');
myDiv.click(function()
{
$('p').toggle('slow');
});
});
```

How do I make DOM elements fade in and out of the page?

For a more sophisticated fading effect, use **$.fadeIn()/$.fadeOut()/$.fadeToggle()**. You use these methods the same way as you used $.show()/$.hide()/$.toggle() in the previous examples.

Here's the paragraph from the previous code sample toggling between visible and hidden using $.fadeToggle().

```
$(document).ready(function()
{
var myDiv = $('#myDiv');
myDiv.click(function()
```

```
{
$('p').fadeToggle('slow');
});
});
```

The effect is very similar to that of $.toggle('slow'). Just try it out.

How do I make DOM elements slide up and down with jQuery?

You can easily make DOM elements appear and disappear with a sliding effect using **$.slideUp()/$.slideDown()/$.slideToggle()**. You implement these methods similarly to the previous ones.

For example, let's say you want to slide the paragraph element from the previous example up and down as the user clicks inside the div element. Here's how you would accomplish this with **$.slideToggle()**.

```
$(document).ready(function()
{
var myDiv = $('#myDiv');
myDiv.click(function()
{
$('p').slideToggle('slow');
});
});
```

Now keep clicking inside the div element and see the paragraph inside the div sliding up and down.

I recommend you experiment with the code samples above as much as you can before moving on with the next challenge.

Try out: sliding menu

Here we are at our jQuery try out exercise. You will build a simple sliding menu similar to what you might have already seen on many websites.

The user moves the mouse over a main menu item and a list of sub-menu items slides down. As the user moves the mouse away from the menu item, its sub-menu items

slide back up away from view.

Let's start from the HTML page:

```
<!DOCTYPE html>
<html>
<head>
<title>Lesson 20: jQuery Sliding Menu</title>

<script type="text/javascript" src="http://code.jquery.com/jquery-
1.7.min.js"></script>

<script type="text/javascript" src="lesson20_ex.js"></script>

</head>
<body>
<h1>Lesson 20: jQuery Sliding Menu</h1>

<div id="myDiv">

<ul class="outerList">
<li><a href="#">Menu Item 1</a></li>
<li><a href="#">Menu Item 2 ↓</a>

<div>
<ul class="innerList">
<li><a href="#">Sub-Menu Item 1</a></li>
<li><a href="#">Sub-Menu Item 2</a></li>
<li><a href="#">Sub-Menu Item 3</a></li>
</ul>
</div>

</li>
<li><a href="#">Menu Item 3</a></li>
</ul>

</div>

</body>
</html>
```

The HTML page above has a reference to a CDN-served copy of jQuery and to an external JavaScript file in the <head> section.

The <body> section has a div element with an id of *myDiv* that contains a list with 3 main menu items and 3 sub-menu items nested inside the second main menu item. The sub-menu items are contained inside a div element.

Style your menu so that the main menu items are horizontally lined up and the nested sub-menu items are vertically stacked under their parent li element (take a peek at the example page if you like).

We want the div element that wraps the sub-menu items to slide down when the user moves the mouse over its parent li element.

Prepare your *lesson24_ex.js* file and write the following code.

```
$(document).ready(function()
```

```
{
//retrieve the menu subitems (div element child of the list
element)
//with the powerful jQuery selectors
//and store them in a variable
var subItems = $('ul li div');
//retrieve the main menu items that
//have the retrieved subitems as children.
//Notice the handy .has() method:
var mainItems = $('ul li').has(subItems);
//Hide all subitems on page load
subItems.hide();
//Attach the .hover() function to the main
//menu items:
$(mainItems).hover(function()
{
//Apply .slideToggle() to the sub-menu
subItems.slideToggle('fast');
});
});
```

Show example

Save your work and preview it in a browser. You should see something like the page indicated by following the example link above, only reflecting your own CSS styles.

Move your mouse over the second menu item and if all goes as expected you'll see the sub-menu sliding down. It's taken us just a few lines of code to accomplish an animation effect that in raw JavaScript would have made us sweat quite a bit.

Summary

You've achieved a lot in this lesson. You know how to use several jQuery functions to manipulate styles. Also, you know how to add fancy effects to your web pages with very little code. Finally, you put your new knowledge to the test by building a core website component - a horizontal sliding menu - using jQuery.

I still advise you to pop over to the jQuery website for a detailed coverage of everything this fabulous library has to offer.

Our next lesson revisits a topic we covered back in lesson 18, get ready for **AJAX done**

the jQuery way.

Lesson 21: Easy AJAX Calls with jQuery

By Maria Antonietta Perna

jQuery provides a rich set of handy methods you can use to Ajaxify your web pages.

Back in lesson 18 you used AJAX by dealing directly with the XMLHttpRequest object. You also performed some feature testing for Internet Explorer browsers that didn't support the XMLHttpRequest object.

jQuery provides **wrapper methods** that shield you from the inner mechanisms of an AJAX request.

In this lesson, you will learn how to use:

- **$.load()** to request content from an HTML page;
- the shorthand **$.get()** method;
- the shorthand **$.post()** method;
- the full-blown **$.ajax()** method.

To follow along with the examples in this lesson, you need to have access to a server, just as you did back in lesson 18.

How do I load HTML content with jQuery AJAX?

jQuery offers a very simple approach to loading HTML content on your web page asynchronously. Just use the **$.load()** function and you're done.

If you use $.load(htmlPageUrl) passing only the URL of the HTML document as argument, the entire content of the HTML document is loaded into the calling page.

Instead, I like to use $.load(htmlPageUrl fragmentIdentifier), which exactly targets the bit of content I intend to retrieve.

Here's how this is done in practice.

Prepare an HTML page containing a div element with an id of *content* and a paragraph element with some dummy content. Save it as *content.html* (or anything you like), and upload it to your server. This document provides the content to retrieve using AJAX.

Prepare a second HTML page containing a link element, a div element with an id of *result*, a reference to the jQuery library in the <head> section, and enclosing <script> tags in the <body> section for your own jQuery-powered JavaScript code.

```
<!DOCTYPE html>
<html>
```



```
<head>
<title>Lesson 21: Easy AJAX Calls with jQuery</title>
<script type="text/javascript" src="http://code.jquery.com/jquery-
1.7.min.js"></script>
</head>
<body>
<h1>Lesson 21: Easy AJAX Calls with jQuery load()</h1>

<p><a href="#">Click here to fetch HTML content</a></p>

<div id="result">

</div>

<script type="text/javascript">

//JavaScript AJAX code here

</script>

</body>
</html>
```

For these simple demos, we're going to embed our JavaScript code inside the HTML page. However, keep in mind that, if you write code for a website, it's highly recommended that you use external JavaScript files.

When the user clicks on the link on your HTML page, an AJAX request will be made to *content.html* targeting the text inside the div element with an id of *content*. This text is dynamically inserted in the div with an id of *result* in the calling page.

To achieve this, enter the following code snippet inside the enclosing <script> tags:

```
$(document).ready(function()
{
//Attach a handler to the click event
//of the link on the page:
$('a').click(function()
{
//Target the div with id of result
//and load the content from the specified url
//and the specified div element into it:
$('#result').load('content.html #content');
});
});
```

Show example

Save your work and preview it in a browser. You should see something like the page indicated by following the example link above.

Click the link, and if all goes well, the dummy text from *content.html* is loaded inside the div element on the page. The operation takes place asynchronously without a full page refresh.

How do I use $.get()?

As nice and simple as $.load() is, it can't perform all types of content requests. A more flexible approach is offered by the **$.get() method**.

You can use $.get() to load data from the server with a **GET HTTP request** - that is, via a **query string** (see **Lesson 10: Passing variables in a URL** in the **PHP tutorial on HTML.net** for a great introduction to query strings).

$.get(url, {dataKey1 : 'dataValue1', dataKey2 : 'dataValue2'}, optionalSuccessFunction) takes in 3 arguments:

1. the**URL where the data you want to retrieve is stored**;
2. optionally, **some data**, if you want to send data to the server with the request. This is done using either a string or what is called **object literal or map**, that is, comma-separated key:value pairs inside curly braces. For instance: **{'First Name' : 'John', 'Last Name' : 'Smith'}** sends the First Name and Last Name values to the server together with the AJAX GET request;
3. and a **function that deals with the returned data if you want to display or process the successful response in any way.**

Let's see how to use jQuery .get() to retrieve the XML document you used back in lesson 18.

The HTML page that makes the AJAX request remains unchanged from the previous example. Rewrite the JavaScript code as follows:

```
$(document).ready(function()
{
//Store the URL value in a variable
var url = "content.xml";

/************************************/

//Package the result-handling code
//in its own function: it's more readable
function processData(data)
{
//This variable will hold the result
```

```
//converted into a string for display
var resultStr = "";
//use jQuery .find() to extract the language
//element from the returned data
//and store it in an array
var items = $(data).find('language');
//loop over each language item with
//jQuery .each() function
$(items).each(function(i)
{
//extract the text of each language item and
//add it to the resultStr variable with a line break.
//Notice the use of $(this)
//to refer to the item currently being
//inspected by the loop
resultStr += $(this).text() + '<br />';
//add the final string result to div element
//with the id of result using .html()
$('#result').html(resultStr);
});
}

/***************************************/

//Attach a click handler to the link element:
//when the user clicks on the link, the AJAX
//request is sent to the server:
$('a').click(function()
{
//use $.get() passing the url variable and
//the name of the result-handling function
//as arguments:
$.get(url, processData);
});
});
```

Show example

Save all your files on the server and click on the link. If all goes well, you should see a list of programming languages being displayed on the page without a full page refresh.

In the example above, you came across **jQuery .find()** and **jQuery .each()**.

$.find() is used to find a DOM element's descendants or children. In the example above, you used it to find all children called *language* of the root XML element contained in the variable called *data*. Further details on $.find() can be accessed on http://api.jquery.com/find/.

$.each(index) is a for ... loop done the jQuery way. It's extremely concise and efficient. Notice the use of **$(this)** inside the $.each() function block. This is a snappy way of referring to the item the loop is currently processing: in our example above $(this) refers to one of the *language items* in the *items array*.

More details on $.each() can be found on http://api.jquery.com/each/.

How do I use $.post()?

If you want to use POST instead of GET in your AJAX calls, you can use **$.post()**.

Unlike GET requests, POST requests don't use a query string to send data. If you intend to send more than a few bits of data to the sever, or if you intend to send sensitive data, it's recommended you use an HTTP POST request.

The way you implement $.post() is very similar to the way you implemented $.get() in the previous example. I invite you to experiment with it on your own and to visit http://api.jquery.com/jQuery.post/ for more code samples and useful details.

How do I use $.ajax()?

If you need greater flexibility, the full-blown **$.ajax()** function offers a great number of settings.

For instance, let's say you want to retrieve a list of programming languages from the XML document you used in the previous example. You might want to specify the following options:

1. the request must be an **HTTP GET**;
2. the page from which the result is returned **must not be in the browser's cache**;
3. the response returned by the server **is of data-type XML**;
4. the request **is made in html**;
5. there must be a **function that handles the returned result if all goes well**;
6. and, finally, there must be a **function that handles errors** in case the request is not successful.

Use the HTML page and the XML document from the previous example. Also, keep the url variable and the processData() function from the previous exercise - you will use both as the url and the success arguments respectively inside the $.ajax() function. Delete everything else inside the document.ready() function. Just below the processData() function, write the following code:

```
//Package the code that handles
//error message in case the request
//is not successful:
function errorAlert(e, jqxhr)
{
alert("Your request was not successful: " + jqxhr);
}

/************************************/

//Attach a click handler to the
//link element on the page
$('a').click(function()
{
//Prepare the AJAX request that
//will be sent when the user clicks the link:
$.ajax(
{
type: "GET",
cache: false,
url: url,
dataType: "xml",
contentType: "text/html",
success: processData,
error: errorAlert
}); //end of $.ajax
}); //end of click handler
});  //end of $.ready function
```

Save your work and preview it in a browser. The result should be similar to the previous example. If an error occurs, you'll be presented with an alert box. The errorAlert() function has an **e** argument that represents the **type of error**, and an **jqxhr** argument that represents the **request as a jQuery object**.

In case an error occurs, details about the error are automatically contained in the arguments provided and will be displayed in the alert box.

Do you want to test the error catching function? Simply replace `dataType: "xml"` in the $.ajax() function with `dataType: "text/xml"`. Save all your files and run the HTML page. Now, when you click the link, an alert box should pop up displaying a parser error message.

More on $.ajax() can be found on http://api.jquery.com/jQuery.ajax/.

Summary

You got to the end of the lesson, and also to the end of this JavaScript tutorial. Congratulations!

Now you're familiar with the **core JavaScript syntax and objects**. You know how to include the **jQuery library** in your projects to add flair to your web pages and **make AJAX calls** easily and efficiently.

I encourage you to keep experimenting with code samples and to be an active participant in JavaScript forums. Why not starting from the forums on HTML.net? It's easy to register and meet with the real experts in your programming language of choice.

As you might have already guessed, the best way to learn coding is to keep coding ... a lot.

The only thing left is for me to wish you hours of fun with your new friends, JavaScript and jQuery.

