JAVASCRIPT

QuickStart Guide

The Simplified Beginner's Guide To JavaScript

JavaScript® QuickStart Guide

The Simplified Beginner's Guide To JavaScript

ClydeBank Media LLC

www.clydebankmedia.com

Follow Us on Facebook & Twitter

Table of Contents

Your Free Gift – Free Audiobook & ClydeBank Media VIP Membership

CLICK HERE TO REGISTER + GET YOUR FREE AUDIOBOOK

<u>WHAT'S A CLYDEBANK MEDIA VIP MEMBERSHIP?</u>

ClydeBank Media VIP Members receive 100% FREE copies of our newly released books in Kindle, Paperback and Audio format. All we ask for in return is the honest, unbiased feedback on the book.

Tell us what you like and what you didn't like about the book after you have read or listened to it. It's that simple.

<u>WHAT'S THE CATCH?</u>

There's no catch. ClydeBank Media VIP members play a crucial role in ensuring we are meeting the needs of our customers by providing important feedback on our products. This feedback allows us to continually enhance our content to guarantee customers are getting the best product possible.

The feedback from VIP members not only provides us with valuable data, but it also has a direct impact on the buying decisions of future customers.

<u>INSTANT ACCESS TO FREE AUDIOBOOK!</u>

Introduction

HTML is not very smart. It mostly lets people look at text and images and allows them to move to other pages where they will do more of the same. What adds the intelligence to a web page is JavaScript. It makes the website more engaging, effective, and useful by letting pages respond to our visitors when they interact with the content.

This book assumes that we already know how to use HTML to specify web page structure and content. It will be additionally beneficial if we are familiar how pages are styled with CSS, separate from the web page structure. If this is the case then we are ready to add a little behavior to the page and make in more dynamic and interactive with JavaScript. Otherwise, without HTML and CSS, JavaScript will not do us much good. They are viewed as the three fundamental pillars of the web page: structure, presentation and behavior.

We will begin this book with an introduction to JavaScript and programming in general. Step-by-step, we will explore the building blocks of programming logic as they exist in JavaScript. We will efficiently build on these fundamentals to introduce interaction, before we move into slightly more advanced territory with functions and objects. Finally, we will introduce the DOM model and learn how to effectively control the browser and all its content.

What is JavaScript?

JavaScript is the scripting language of the web with the sole purpose of adding interactivity to our pages. In addition to interactivity, modern versions of JavaScript can also be used to load and parse information from external sources or even the website's users. JavaScript is essentially a piece of programming code embedded in the HTML structure of a web page. When the web browser reads this code it activates a built-in interpreter that understands how to decipher this language and process its commands.

Although programming is involved during coding, JavaScript is not a programming language. In conventional web programming languages, like Java or .NET, the code has to be compiled before it is executed. Compiling means that the code has to be first sent to a special program that is run on the server. This program, also known as application server software, translates the code, creates the requested page and/or functionality and serves this back as HTML. Scripting languages, like JavaScript, are note compiled, but rather are interpreted on-the-fly. This means that no special software is involved as the user's own browser runs and executes the code as it is encountered.

> ***Note:*** *JavaScript was created during a time when Java was a very popular language. Other than that, the languages are not related and have almost nothing in common except for basic programming logic.*

The Birth and Growth of JavaScript

Initially, called LiveScript, JavaScript was created by Brendan Eich in 1995 in just 10 days. The first idea behind the language was to improve the Netscape Navigator browser by adding scripting capabilities to the front-end interface. However, the language was so well received that within a year it was reverse-engineered and included in the Internet Explorer browser. By 2000 the language gained a lot of popularity and expanded significantly so it was submitted for standardization to the ECMA Committee.

Once it became standardized, JavaScript continued to grow in adoption and popularity in tandem with the expansion and growth of the World Wide Web and the refinements of the browsing experience. As a matter of fact, Google gave it a big push into the professional spotlight when it released Google Maps in 2005 and showed how JavaScript can be used to create exciting and dynamic interaction.

Implementing JavaScript

Now that we have a general idea as to what JavaScript is, we can start working with this language. As JavaScript code is part of the HTML document, we need to know how to tell browsers to run our scripts. There are two common options available to us when we want to include JavaScript in a web document and in both cases we will use the <script> element. The <script> tag is used when we want to tell the browser where the JavaScript code begins and where it ends within an HTML document. As such, this tag can be included either in the head or the body section of the page.

The first option is to place the code inline within the document structure. To do this we will begin by opening a <script> tag, entering the JavaScript code, and then finish by closing with the </script> tag. We can theoretically leave the document like thisas almost all browsers will assume that the scripting language between the <script> tags is JavaScript by default. Nevertheless, for maximum compatibility we will extend this tag with the type attribute and the text/javascript value in order to instruct the browser how to exactly interpret the code.

```
<script type="text/javascript">
        //A JavaScript comment
</script>
```

The second option is to load the JavaScript code from an external file into our HTML document. For this purpose we can use the <script> element again, but this time in addition to the type attribute we will also include the URL to the external file in the src attribute of the <script> element. The external file must be a text-only file with the .js file extension that contains only pure JavaScript code without any HTML elements or CSS rules. For example, to call the external scripts.js file into our browser we would use the following code:

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

We put JavaScript in an external file and include it in the web page when we like to share the functionalities across our entire web site. Otherwise, if we just need to add some local interactive behavior, we embed the code within the page.

> ***Note:*** *Script files are loaded in the order in which they are placed in the HTML code*

Chapter 1: Fundamental JavaScript Concepts

Generally, when we hear the term programming we immediately think of other people typing an incomprehensible string of letters and numbers. Programming looks like magic beyond the realm of mere mortals. Nevertheless, the concepts in programming are not difficult to grasp as they always have real life applications. JavaScript, although it is not as simple as HTML or CSS, is not an overly complicated language. Unlike other languages, its "grammar" is more or less descriptive and intuitive making it a good fit for a first programming language. Basically, learning JavaScript is like learning a new language, but a new language that is similar to English. Once we learn the new words, and understand how to put them together to form "sentences" we'll be good to go.

Syntax

Every language has its own set of characters and words that go along with a set of rules as to how to arrange these characters and words together into a well-formed sentence. These rules are also known as the language syntax and it is the syntax that holds the language together and gives it meaning.

Before start with some examples of JavaScript syntax, let us first set up the environment for JavaScript. As discussed previously, JavaScript code is always a part of the HTML code. Therefore, in order to work with JavaScript we will first need to create a basic HTML document. So to start, let us open a text editor (like Notepad) and type in the HTML code for the most basic web page. In addition to the basic HTML tags, we will include a <script> element in the <head> section where we will start placing the JavaScript code.

```
<!doctype html>
<head>
<title>First Steps in JavaScript</title>
<script type="text/javascript">

</script>
</head>
<body>
```

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

Let us save this document as firststeps.html. If we are using Notepad, we have to remember to change the Save as Type field to ‘All files’.

Statements

To express ourselves in everyday common language we use sentences as the basic form of communication. Similarly, in JavaScript we also form sentences to express our intentions which are more formally called statements. A JavaScript sentence is the basic unit of communication, usually representing a single step in the program. And just like we put sentences together to express an opinion, we combine statements together to create a program.

Let us look at a simple JavaScript statement and see what it does. Between the opening and closing <script> tag of the html document place the following text:

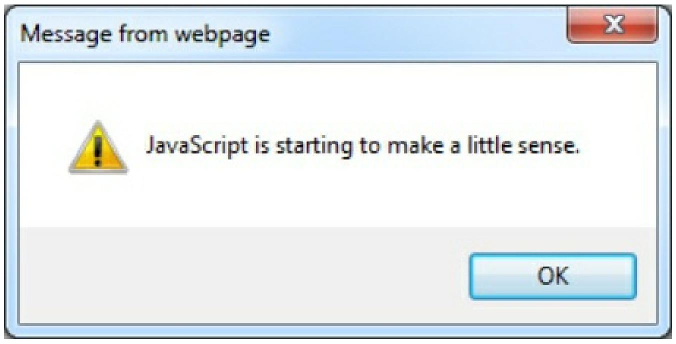
```
alert("JavaScript is starting to make a little sense.");
```

In further examples we will not show the complete HTML code unless it is necessary, but for initial reference your document should look like the following:

```
<!doctype html>
<head>
<title>First Steps in JavaScript</title>
<script type="text/javascript">
        alert("JavaScript is starting to make a little sense.");
</script>
</head>
<body>

</body>
</html>
```

We can save the firsteps.html document and open it in a web browser. Once the page opens, we will get an alert window with the message *"JavaScript is* *starting to make a little sense".*


Message from webpage

JavaScript is starting to make a little sense.

Image 1. Alert window

Now that we know what the effect is, let us go back to the JavaScript statement and interpret it into common language so it makes more sense.

```
alert("JavaScript is starting to make a little sense.");
```

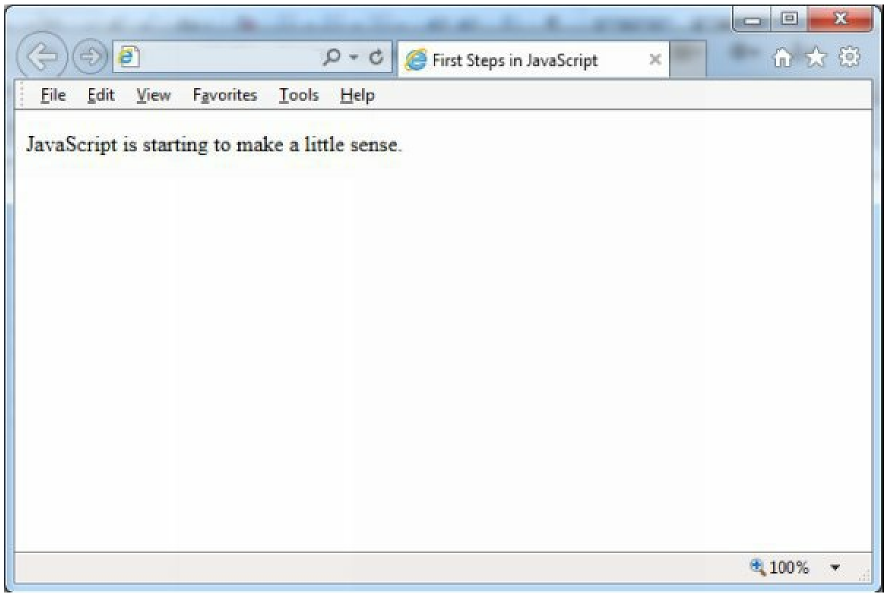
JavaScript statements are instructions whichare executed by the web browser. The statement starts with a command, presented by keyword. The keyword identifies the action that needs to be performed. In this case the keyword alert makes the web browser open a dialog box and display a message. If we just had the statement alert(); the dialog box would have been empty, however in our case the statement consists of a specific input, the actual message text, also known as an argument. Finally, just like every sentence ends with a period, a JavaScript sentence ends with a semicolon. The semicolon makes it clear that the statement is over and once the interpreter executes it, it should move on to the next item.

Now we are ready to translate the JavaScript statement. Its plain English interpretation would be, "Open a dialog box and display the text *JavaScript is starting to make a little sense' in that box.*"

> ***Note:*** *When passing text arguments we can use either double quote marks ("sense") or single quote marks ('sense') present the text.*

Before we move on, let us look at another JavaScript statement. In the <script> element replace the previous code with the following and preview it in a web browser to see the results:

```
document.write("<p>JavaScript is starting to make a little sense.
</p>");
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

JavaScript is starting to make a little sense.

Image 2. Example of a document.write statement

What we can see from the results is that the previously empty document, now has one paragraph of text. Following the previous interpretation of how JavaScript works and from the web browser results we can correctly assume that the document.write keyword commands the browser to write directly onto the web page. Similar to alert() , it writes whatever is placed between the opening and closing brackets.

Variables

One of the fundamental aspect of JavaScript, and any programming language in general, is the concept of variables. A variable is a way to declare and store information which can later be used. This information can vary with the circumstances and hence the name variable.

Let us look at the following statement.

```
var name = "Martin";
```

In plain language this is the same as saying "My name is Martin". The keyword var is JavaScript speak for "create a variable", or in a programming dialect, "declare a variable". What follows is the name of the variable whichcan be anything we choose with certain limits. Assigning a value to the variable is done with the = sign, which is not immediately necessary, as this can happen later. We can declare an undefined variable in one statement, and assign it a value in a later statement. For example:

```
var name;
name = "Martin";
```

As mentioned previously, variable names can be anything, like name , abc , R2D2 , with a few rules. Variable names can contain letters, numbers, dollar sign ($), or lower line (_), other special characters are not allowed. Furthermore, a variable name cannot begin with a number,any other allowed value is acceptable. Finally, variable names are case-sensitive, meaning that the interpreter in the web browser makes a distinction between uppercase and lowercase letters, making 'score' different from 'Score' .

> ***Note:*** *Although we can use almost anything for a variable name, it is wise to use names which are meaningful as this will help us and other programmers to better understand the written code.*

Variable Types

Based on the type of data, variables come in different flavors. The three most basic types are number, string, and boolean.

A **number** variable is represented by a numeric character. This variable can accept whole integers, negative integers and fractional integers. Numbers are frequently used in calculations, hence our number variables are often included in mathematical operations. The following statement declares a variable named age and assigns it a value of 35.

```
var age = 35;
```

A **string** variable is used to represent any series of letters like words or sentences. Strings are represented as a series of characters enclosed within quotation marks, with the quotation marks signaling to the interpreter that what follows is a string variable. JavaScript allows us to use both the double quotes (") or the single quote (') marks, but we have to be mindful to use the same type of quotation mark.

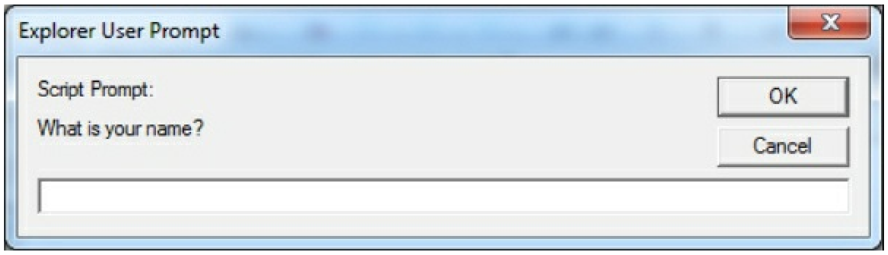
```
var location = "California";
```

A **boolean** variable is rather simple as it can accept only one of two values: true or false. This variable is used when we create JavaScript programs that we want to intelligently react to user actions. We will address them in the next chapter when we start discussing more complex programs.

User Variables

JavaScript would not be fun if it didn't allow us to share our thoughts and create or alter the variables directly. One of the simplest ways to "give" our input is to use the prompt() command.

```
var name = prompt ("What is your name?", "")
document.write(name);
```



Script Prompt:
What is your name?

Image 3. A custom prompt dialog box

The result of the prompt() command is a dialog box. Instead of just displaying a message like the alert dialog box, the prompt dialog box can also receive an answer. Hence, in the syntax for a prompt dialog box it is necessary to provide two arguments between the parentheses separated with a coma. The first argument is the prompt text that is displayed in the box, while the second argument is the default value for the text box, and consequently, the variable.

In the example above, the prompt text displayed in the box will be *'What is your name?'* and the default value presented in the box will be empty, as there is obviously no content between the quotation marks. Once we type in something in the box and either click OK or press the Enter key, the variable receives the value that was entered in the field. Consequently, the name will be displayed on the web page. Otherwise, if we click on Cancel, press the Esc key or close the prompt box, the returned value would be empty and there would be no text on the screen.

Note:Instead of prompt() we can also use the more formal window.prompt command.

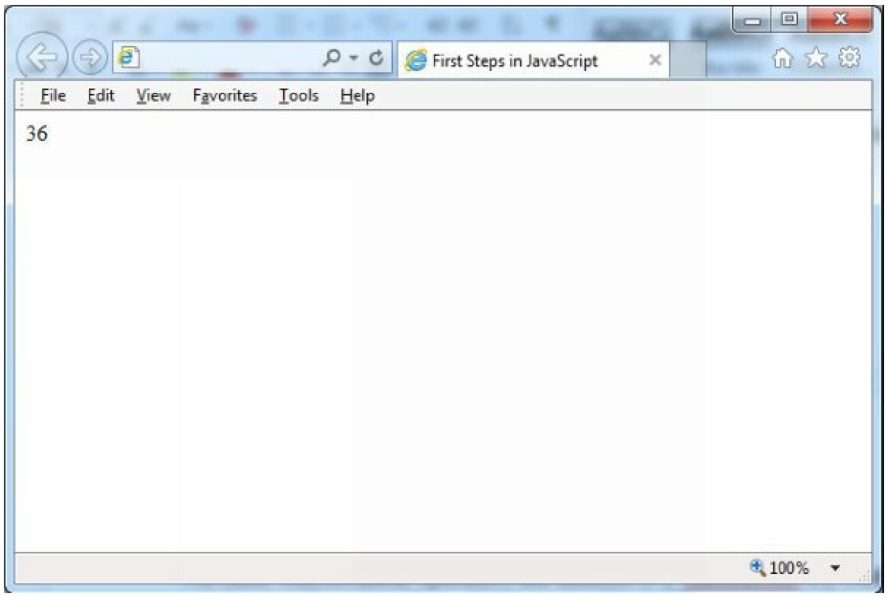
Operators

Storing information in a variable is a first step, the beauty of programming is the ability to manipulate this information in many creative ways. For this, JavaScript provides different operators that allow us to modify data. An operator, represented by a symbol or a word, is something that can change one or more values into something else. The type of operators available are different based on the data type.

Mathematical Operators

The basic mathematical operators like addition (+), substraction (-), multiplication (*) and division (/) are readily available in JavaScript. They can be used in independent statements or used when declaring variables. For example, by "operating" with the variables currentYear and yearofBirth, we can determine the value of the variable age.

```
var currentYear = 2015;
var yearofBirth = 1979;
var age = currentYear - yearofBirth;
document.write(age);
```



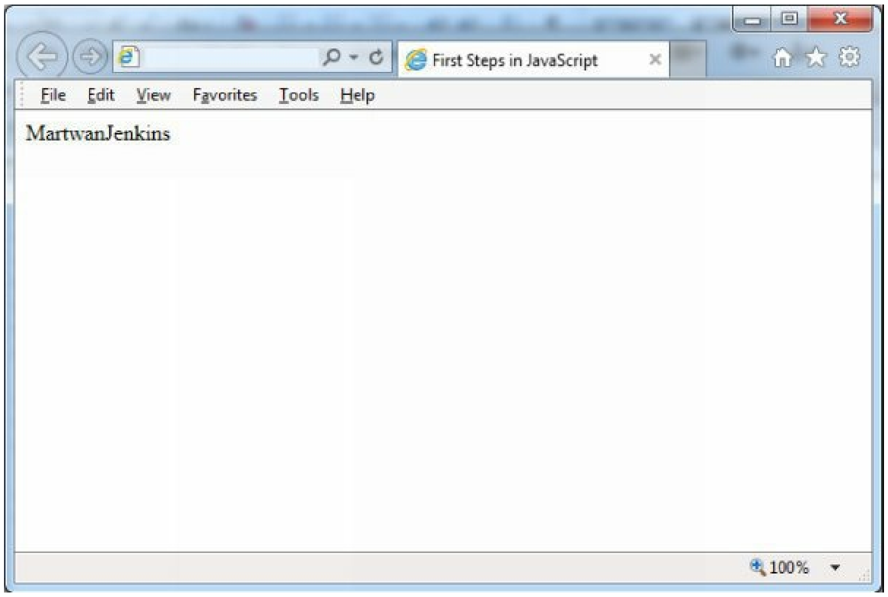
```
36
```

Image 4. Using variables to calculate age

> ***Note:*** *"Calling" a variable to be presented on a web page is easy. Simply use the document.write() command.*

Mathematical operators, specifically the addition operator, can be used to combine two or more strings. This process of combining strings is called concatenation. In the following example:

```
var firstName = "Martwan";
var lastName = "Jenkins";
var fullName = firstName+lastName;
document.write(fullName);
```

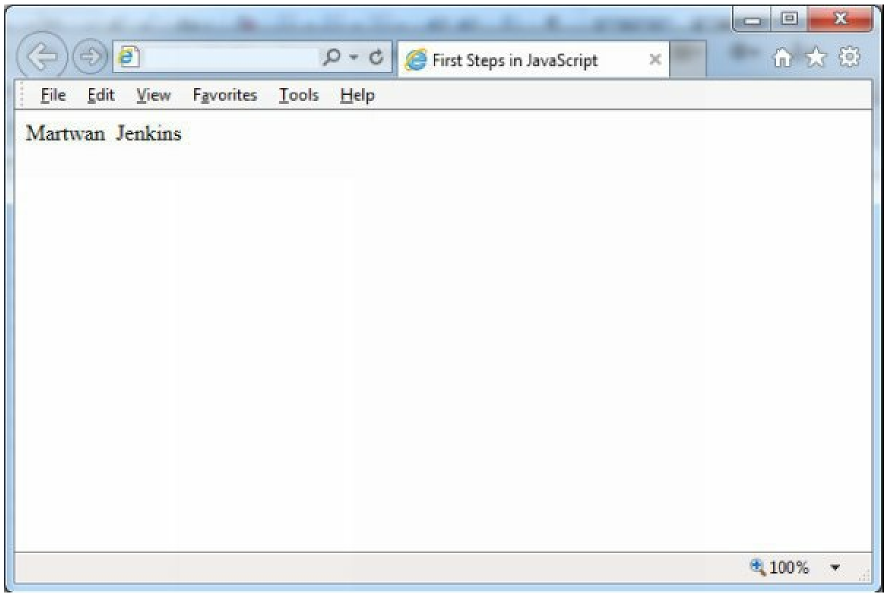


```
MartwanJenkins
```

Image 5. Concatenating strings into a full name

The value for fullName will end up being MartwanJenkins . To make sure that everything is in its proper form we need to include the empty space as a string in quotation marks. For example, we can use the following declaration:

```
var fullName = firstName+" "+lastName;
```

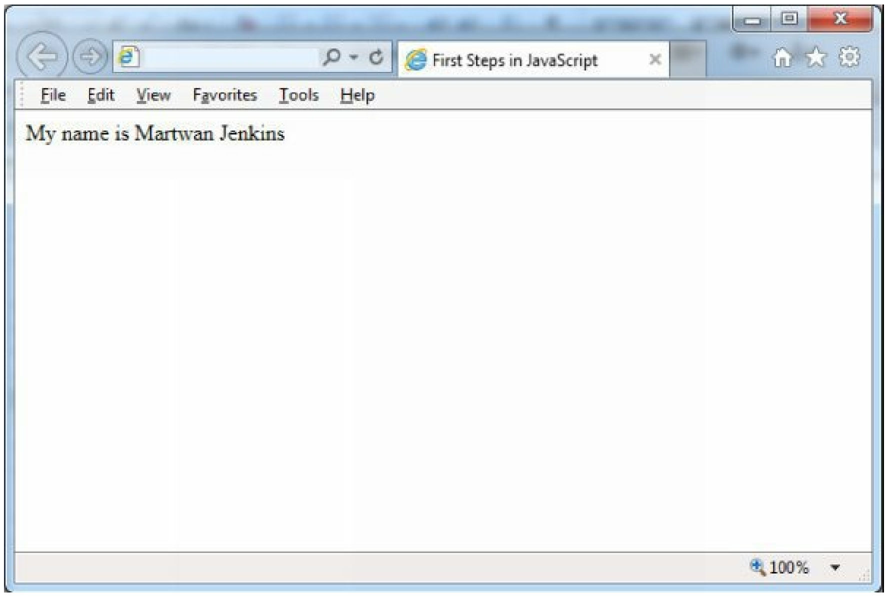

First Steps in JavaScript
File Edit View Favorites Tools Help
100%

Martwan Jenkins

Image 6. Concatenating strings with spaces

We can see that operators are also useful when we want to join text and/or combine variables. As a matter of fact, we can use this to construct more logical sentences. For example, we can combine the "My name is" text with a value from a calculated variable.

```
var firstName = "Martwan";
var lastName = "Jenkins";
var fullName = firstName+" "+lastName;
document.write("My name is " +fullName);
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
My name is Martwan Jenkins
```

Image 7. Concatenating a full sentence

> ***Note:*** *When performing several mathematical operations in one statement, the rules of precedence apply.*

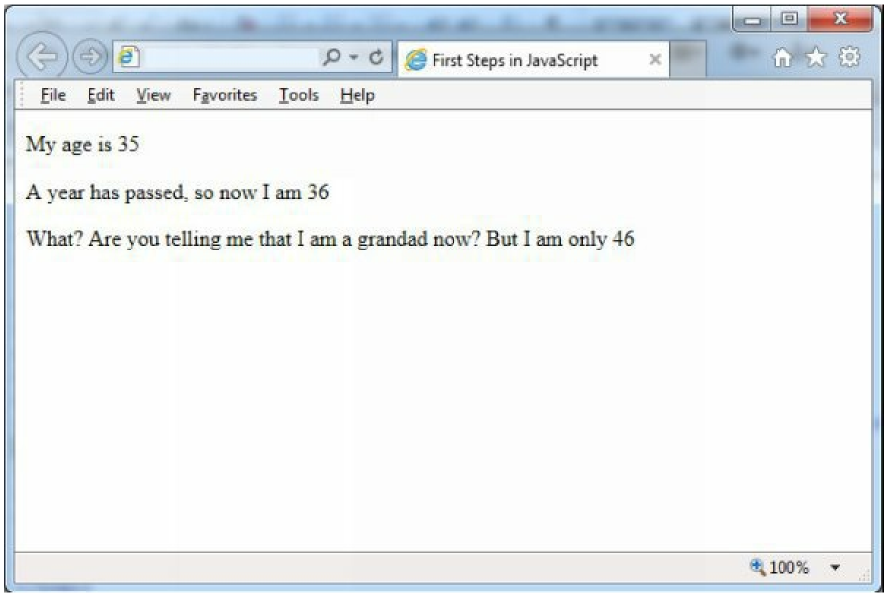
Assignment Operators

Just like things change in real life, so do variables within a JavaScript program. And to change variables within JavaScript we will use assignment operators. We are already familiar with the fundamental assignment operator, the equal sign (=), which is used to give an initial or a new value to a variable. There are other assignment operators that also change the value of a variable, but they do this in a slightly different way.

For example, as the year passes we grow older and our age incrementally changes by one. To make this change in JavaScript there are several different approaches we can take, all with the same results. To play around with the possibilities of changing variables, we can try the following code where after the age variable changes its value is displayed in the browser:

```
var age = 35;
document.write("<p>My age is "+age+"</p>");
age = age + 1;
document.write("<p>A year has passed, so now I am " +age+"</p>");
age += 10;
```

```
document.write("<p>What? Are you telling me that I am a grandad
now? But I am only " +age+"</p>");
```



```
My age is 35

A year has passed, so now I am 36

What? Are you telling me that I am a grandad now? But I am only 46
```

Image 8. Changing variables with assignment operators

While at first these operations might appear slightly confusing, they are still logical if viewed from the programming angle. For example, if we read the statement age=age+1 backwards what happens is that the value of 1 is added to the current age of 35 which would make 36 the new value of age. Additionally, we can use a complex assign operator such as (+=) in the statement age+=10 , to increase the value of the variable by 10. We can also use the same logic to other operations like subtraction, division and multiplication.

```
age-=5;          is the same as          age = age - 5;
age*= 5;          is the same as          age = age *
5;
age/=5;          is the same as          age = age / 5;
```

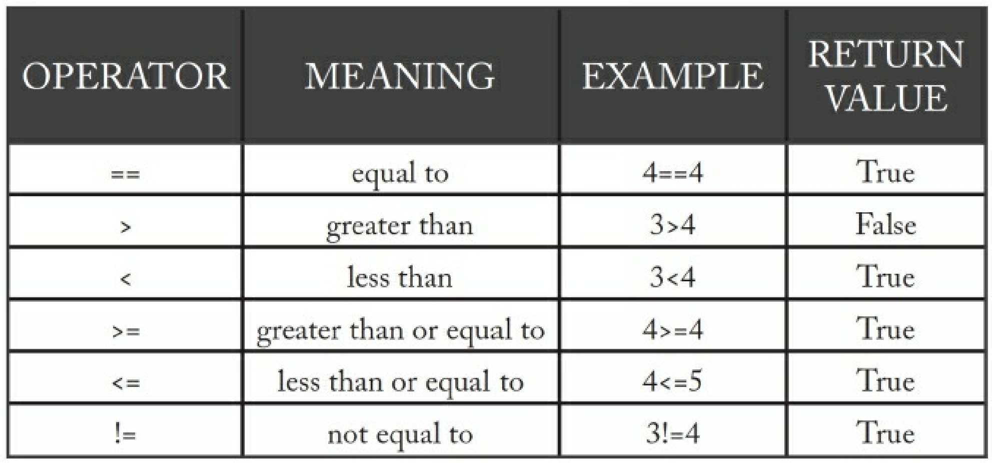
Additionally, when we want to increase/decrease the value of the variable by 1, we can also use the following assignment operators:

```
age++;          is the same as          age = age + 1;
age--;          is the same as          age = age - 1;
```

Comparison Operators

Like the name suggests comparison operators are used to compare two values. After the comparison is made a value of either true or false is returned depending on whether the comparison was exact or not. Comparison operators are mostly used when evaluation conditions in loops or conditional statements, discussed in the next chapter.

The following table summarizes the most common comparison operators.



OPERATOR	MEANING	EXAMPLE	RETURN VALUE
==	equal to	4==4	True
>	greater than	3>4	False
<	less than	3<4	True
>=	greater than or equal to	4>=4	True
<=	less than or equal to	4<=5	True
!=	not equal to	3!=4	True

Table 1. Comparison operators

Logical Operators

Logical operators allow us to compare two or more conditional statements. The comparison lets us determine which statement is true so we can proceed accordingly. Just like comparison operators, logical operators return a value of either true or false, depending on the values on either side of the operator. The three most common logical operators are AND, OR and NOT.

The AND operator is represented by the && code. Only when statements on both sides of the && operator are true, the operator returns a true value. If one or both statements are false, the operator returns a false value. This is in agreement with the rules of common language, for example, the statement *"This jacket has all the American colors, red, white and blue"*, is only true if the jacket has red (true), white (true) and blue (true). If only one of those colors is different (false), then the statement would also be false. In JavaScript syntax, this would look something like the following:

```
(color1 = "red" && color2 = "white" && color3 = "blue")
```

The OR operator is represented by the || code. It returns a true value if statement on one or both sides of the operator are true. Only if both statements are false, the operator would return a false value. For example, the statement *“My brother will either go to Brazil or Thailand for his summer holiday”* would be true if my brother visited one (or maybe both) of those places.

```
(destination1 = "Brazil" || destination2 = "Thailand")
```

The NOT operator, represented by the ! code, is used to “claim” that the opposite statement is true. So, the operator will return a true value for any other condition than what is explicitly stated. For example, the statement *“Brian does not weight 200 pounds”* is true for any other value of weight except 200.

```
!(weight == 200)
```

We will learn the more practical aspects of logical operators in the next chapter where we focus on conditional statements and loops.

Arrays

The variables we examined up until this point can remember only a single piece of information at a time. However, more than often we need to keep track of multiple values like the months of the year or a shopping list. In such a case, JavaScript provides a nice way to remember multiple values under one variable, called array. Let us start by defining an array variable that will hold the first six months of the year.

```
var months = ["January", "February", "March", "April", "May",
"June"];
```

When declaring an array variable we first use the var keyword followed by the name of the variable just like with regular variables. As the array contains multiple items, we will provide a value for each item between opening and closing square brackets, [] . Like with normal variables if the value of the item is a string we will place it within quotation marks. Otherwise, if the value is a number or possibly another variable we will not use any quotation characters. As an example, we can declare the following array of random items:

```
var randomItems = ["ball", -0.5, "black pony", 333, true]
```

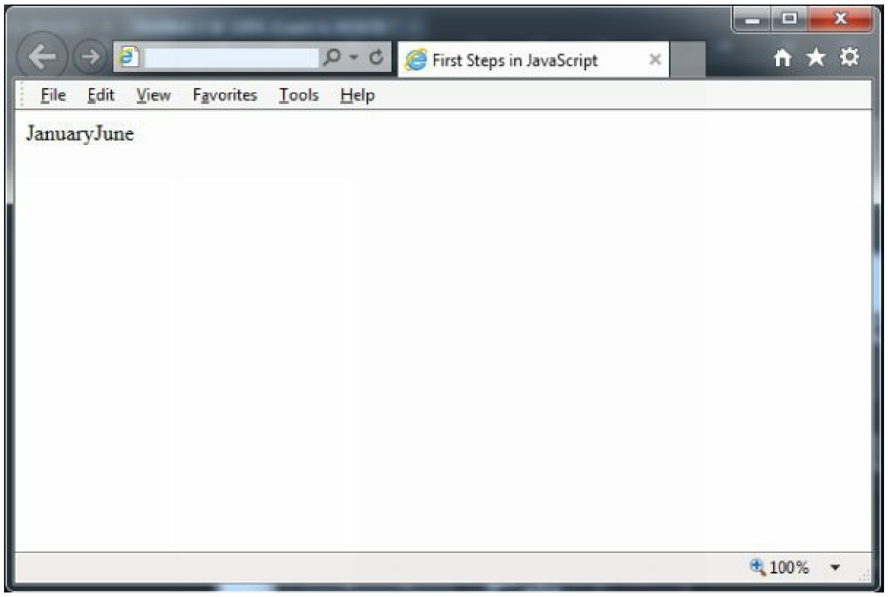
It is also possible to define an empty array. We will use the same syntax, but leave out the values for any item.

```
var hole = [];
```

Working With Array Items

Accessing array items is not the same as accessing common variables. Since an array holds multiple values it is not sufficient just to use the name of the array variable, we also need to indicate its location within the array. The position of each item in an array is indicated by a number called an index. Therefore, to access an array item we have to use the array variable name and the index number. For example, if we want to display the first and the last item from our months variable we will use the array name followed by a number enclosed in square brackets.

```
var months = ["January", "February", "March", "April", "May",
"June"];
document.write(months[0]);
document.write(months[5]);
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
JanuaryJune
```

Image 9. Example for defining and displaying array results

Note: *JavaScript numbers each array item automatically. The index value for the first item is always 0, the index value for the second item is always 2, and so on.*

We will use the index position in the case when we want to change the value of a specific array item. For example, let us say that in our months array we want to replace the name of the months with their shorthand. We can change each variable with the following statements:

```
months[0] = "Jan";
months[1] = "Feb";
months[2] = "Mar";
months[3] = "Apr";
months[4] = "May";
months[5] = "Jun";
```

Managing Array Elements

If we know the index value of the last array item, we can simply add more items by assigning values to subsequent items. For example, the following statement will add one more item in the array under the index value of 6.

```
months[6] = "Jul";
```

However, while programming we are almost never required to actually remember such trivialities such as the last index value. Instead, to add one or more elements to the end of an array we can use the keyword push . To add the final five months to our months array, we will combine the push keyword with the array name to form the following statement:

```
months.push("Aug", "Sep", "Oct", "Nov", "Dec", "Jan");
```

Well, it seems that we went over and added an extra month. To remove the last element of an array we will use the pop keyword. Like previously, we will combine the pop keyword with the array name to form the following statement:

```
months.pop();
```

This statement will delete the last array element, "Jan" , leaving us with an array containing all the months of the year.

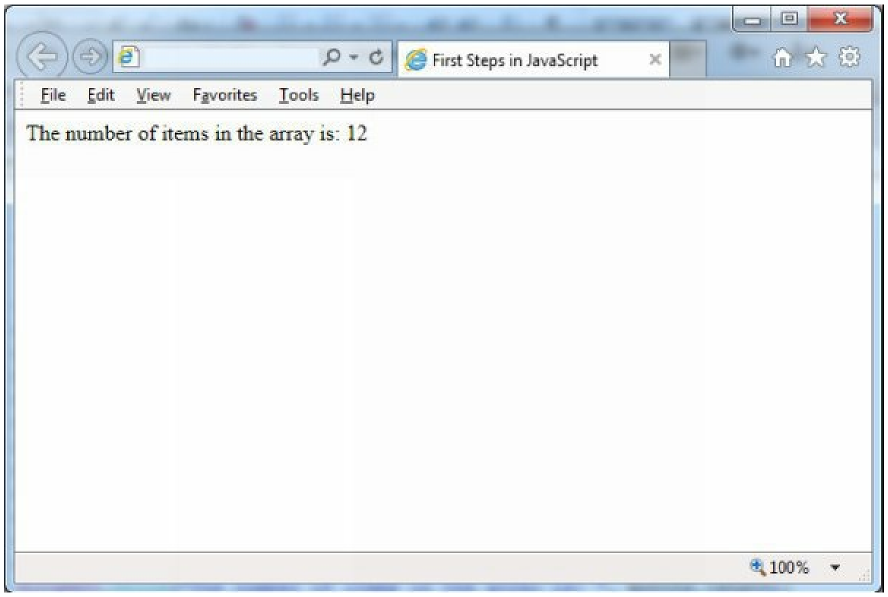
When we want to add an element to the beginning of an array instead of the end, we can use the unshift keyword. Alternatively, when we want to remove an element from the beginning of an array we can use the shift keyword. The following two statements will add and subsequently remove a bogus first month to our array.

```
months.unshift("Sext");
months.shift();
```

If at any time we want to check the number of elements that are contained within an array, we can initiate the length property for the monthsarray, months.length , and print it with the document.write statement.

To review all the different aspects we have covered our complete code will be:

```
var months = ["January", "February", "March", "April", "May",
"June"];
months[0] = "Jan";
months[1] = "Feb";
months[2] = "Mar";
months[3] = "Apr";
months[4] = "May";
months[5] = "Jun";
months[6] = "Jul";
months.push("Aug", "Sep", "Oct", "Nov", "Dec", "Jan");
months.pop();
months.unshift("Sext");
months.shift();
document.write("The number of items in the array is: ",
months.length);
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
The number of items in the array is: 12
```

Image 10. Manipulating and displaying the array length

Comments

From working with HTML and CSS we should already be familiar with comments. Basically, commenting is a way to tell the browser to ignore segments of the code. The purpose of comments is to leave notes about our code either for us or whoever is going to read the code after us. As JavaScript can be more difficult to understand when it is revised on a later date, commenting is even more important.

JavaScript recognizes two different approaches to commenting, marking a single line as a comments and marking multiple lines as comments. Regardless of the approach, lines which are commented will not be executed by the interpreter.

Single line comments are created by two forward slashes. When the interpreter finds two forward slashes it will ignore everything that follows until the end of the line.

```
//this is a single line comment
```

It is also possible to write a line that is part code and part comment. In the following example, after the variable year is declared, the rest of the line will be treated as a comment.

```
var year = 2015; //this is most likely the current year
```

When we need more than one line of comments we can use an alternative approach. Multiline comments in JavaScript are the same as multiline comments in CSS. The comments have a beginning, initiated by a forward slash and an asterisk (/*) and they have an end initiated by an asterisk and a forward slash (*/). Anything between these opening and closing comment signifiers will always ignored by the interpreter.

```
/*This is a comment that
is written in exactly
three lines */
```

Chapter 2: Conditional & Loop Statements

Creating a variable and storing a value in the variable is not really an accomplishment, neither is changing this variable with simple operators. What we need to do to start making things interesting is to make our programs react to our actions, make them more intelligent And we can do this by using conditional statements and loops.

Conditional Statements

We make a myriad of choices during our everyday activities: *"Should I get out of bed?", "What should I eat?", "Where should I go for coffee?"*. These choices depend on the current situation as our decisions are affected by the surrounding circumstances. In a similar fashion, JavaScript also has decision-making capabilities called conditional statements. Fundamentally, conditional statements are a simple yes or no question with the program doing one thing if the answer is yes, and either nothing or another thing if the answer is no.

The most basic conditional statement is the so-called if statement. This statement executes a task only if the answer to the question is true. To understand the syntax of an if statement let us look at the following example:

```
var age = prompt("What is your age?","");
if (age > 30) {
        document.write ("You are not so young anymore.");
}
```

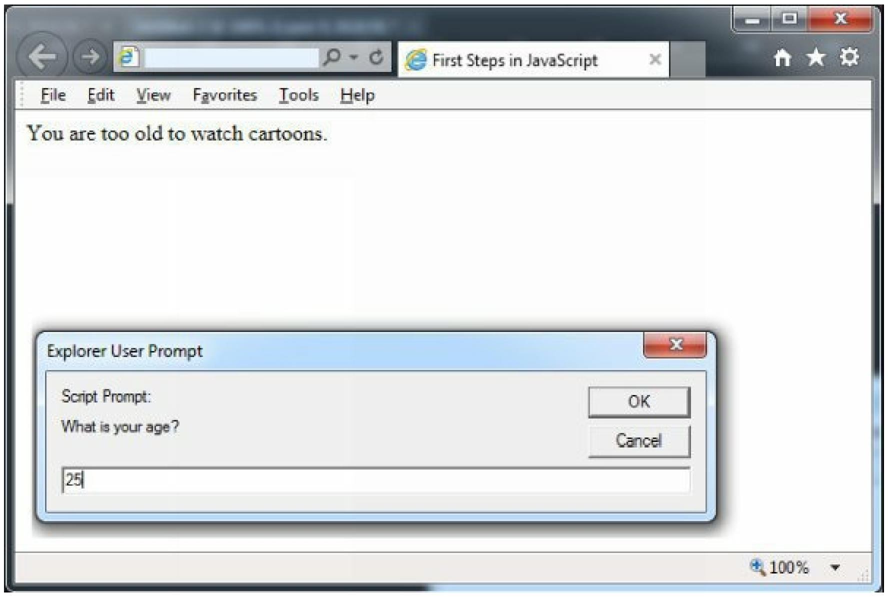
The if statement consists of three parts. The if keyword indicates that what follows is a conditional statement. The parentheses, (), contain the yes/no question, also known as the condition. Finally, the curly braces, {} , contain one or more statements that need to be executed if the answer to the question is positive. Essentially, the { bracemarks the beginning and the } brace marks the end of the code that will be run if the condition is fulfilled.

In our example, the condition is a comparison between two values. With the condition (age>30) we check whether the age variable has a value greater than 30. If the condition is true then the statement to write the text within the brackets on the web page is executed, otherwise, if false, the interpreter will skip all of the statements within the curly brackets.

Alternative Conditions

In real life when one condition fails we always have an alternative. If the restaurant is out of cake, we can always order a fruit salad for dessert. Similarly, in JavaScript we also want something to happen when the condition is true and when the condition is false. To achieve this we can extend the if statement with an else clause. For example:

```
var age = prompt("What is your age?", "");
if (age < 18) {
        document.write("You can enter the web site.");
}
else  {
        document.write("You are too old to watch cartoons.");
}
```


First Steps in JavaScript
File Edit View Favorites Tools Help
Explorer User Prompt
OK
Cancel
100%

```
You are too old to watch cartoons.
```

```
Script Prompt:
What is your age?
25
```

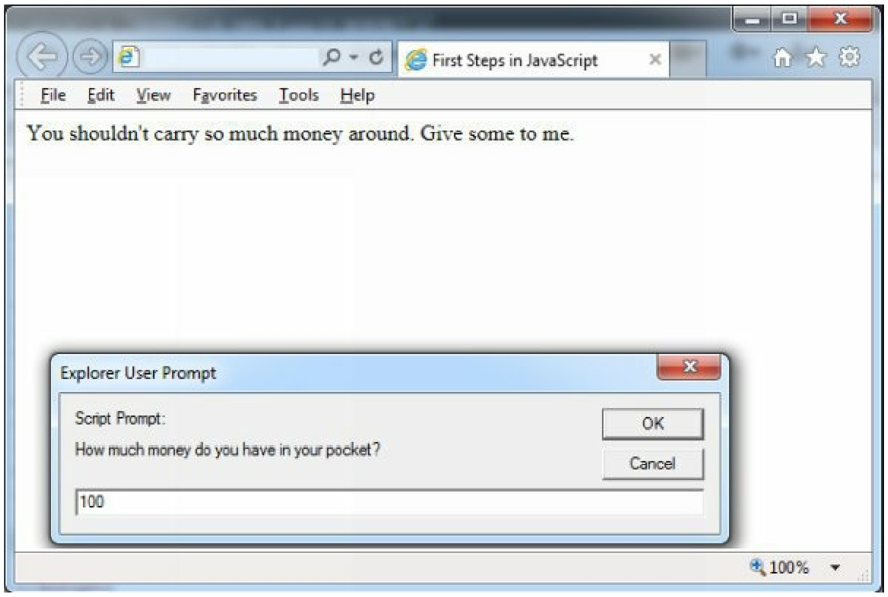
Image 11. Using a conditional statement to respond to users' input

In the example above the user is asked to enter his/her age. Then the if statement evaluates whether the value of the age variable is less than 18. If this is true , the text *"You can enter the web site"*is displayed on the page. If this condition is false , and the user has entered a value that is equal to or greater than 18, than the else clause is initiated and the text *"You are too old to watch cartoons"* is displayed on the web page.

To initiate an else clause we simply add it as a keyword after the closing brace of the conditional statement. The statement(s) that we want to execute are also placed in braces {} , as we have the option to add as many lines of code as necessary.

More than often there are more than two possible outcomes to a situation. When this is the case, JavaScript lets us use cascading else if statements to offer solutions to multiple alternatives. We start with an if statement for the first option, and then add one or more else if statements to trigger additional options. Like previously the else clause is used in the end as the last alternative?

```
var money = prompt("How much money do you have in your pocket?",
"");
if (money < 20) {
          document.write("That is not enough. Go to the ATM to get
some more.");
}
else if (money == 20) {
          document.write("Exactly 20? That's great, buy me a nice
burger meal.");
}
else {
          document.write("You shouldn't carry so much money around.
Give some to me.");
}
```


First Steps in JavaScript
File Edit View Favorites Tools Help
Explorer User Prompt
OK
Cancel
100%

You shouldn't carry so much money around. Give some to me.

Script Prompt:
How much money do you have in your pocket?

100

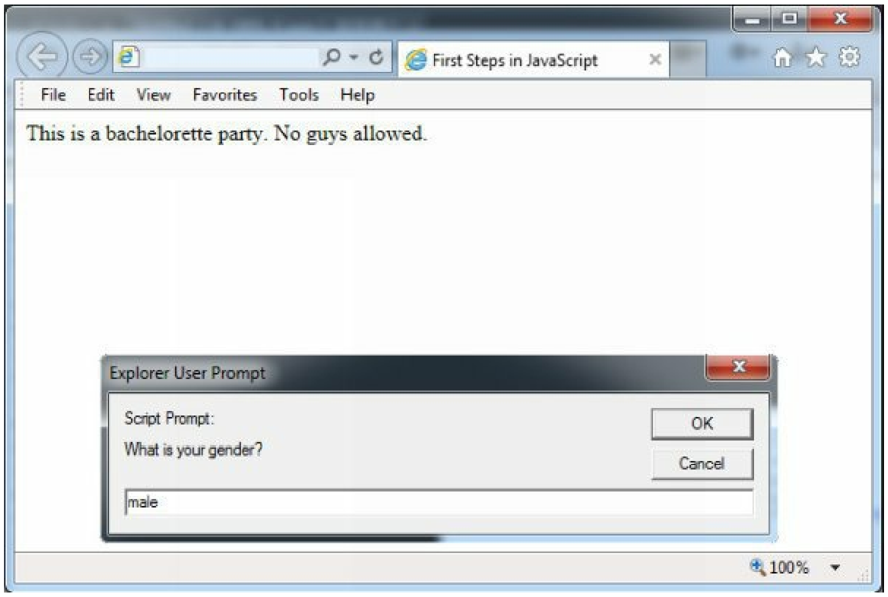
Image 12. Using alternative conditional statements

In the example above, the user is asked to enter the amount of money he/she possesses. Afterwards, the JavaScript program assesses the answer and replies with one of three available responses based on the entered amount.

Multiple Conditions

Now that we have covered all alternative, let us see what happens when there is more than one variable. When we are dealing with different variables we will need to form more complex conditional statements. For example, if we are having a private bachelorette party we must make sure that we don't let people in unless they are female and over 18. Using the logical operators from the previous chapter and the if/else conditional statements we can build more intricate comparisons.

```
var gender = prompt("What is your gender?","");
var age = prompt("How old are you?","");
if ((gender == "female") && (age >= 18)) {
        document.write("Please enter. Hope you brought a nice gift.");
}
else if (gender == "male") {
        document.write("This is a bachelorette party. No guys
allowed.");
}
else {
        document.write("You are probably too young to enter. Come
back in a few years.");
}
```


First Steps in JavaScript
File Edit View Favorites Tools Help
Explorer User Prompt
OK
Cancel
100%

This is a bachelorette party. No guys allowed.

Script Prompt:
What is your gender?

male

Image 13. Checking for multiple conditions

In the example above we are making decisions based on two factors, gender and age. The only positive decision is available when two conditions are true, the person is a female over 18 years of age. In JavaScript, we combine conditions using logical operators, which in this case is the AND operator represented by double ampersands (&&). When using this operator between two conditions within a single conditional statement we summarize the outcome into only one condition.

Loop Statements

During our daily activities to complete a task we have to repeat the same series of steps over and over again. To wash the dishes, we have to wash each dish separately. To climb down the stairs we have to step down one stair at a time. At the checkout counter, we have to take each item out of the cart, scan it and put it back in a bag. Well, in programming repetitive tasks are very easy to do. As a matter of fact, JavaScript is very good at performing repetitive tasks as it has all the necessary tools to do the same thing over and over again. In programming lingo, repeatedly performing the same task is called a loop.

There are several different types of loops. They essentially do the same thing, but the approach is slightly different. In this section we will familiarize ourselves with while loops and for loops.

While Loops

In a while loop, the same code is repeated as long as a certain condition is true. Let us dive into an example that checks our age as we grow every year.

```
var age = 1;
while (age < 18) {
        document.write("Another year has passed. You are now "+age+" years old.<br>");
        age = age + 1;
}
document.write(age+"? Congratulations! You are now an adult.");
```

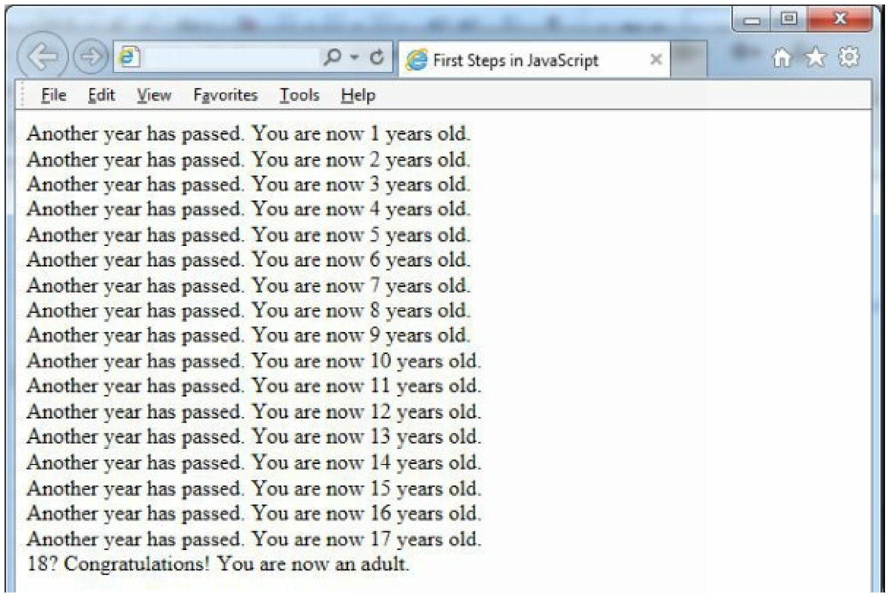
In this example, after declaring the age variable, we introduce a while statement. Following the while keyword we place a condition between parentheses, which in our case is age<18 . If the condition is true, the JavaScript interpreter will run the code that appears between the braces. Unlike a conditional statement, when the closing brace of the while statement is reached, JavaScript does not continue with the rest of the program, but instead it initiates the while statement again.As long as the number stored in the age variable is less than 18, the script will run the document.write() command.

Only when the condition is false, the interpreter will 'exit' the loop and continue parsing the rest of the code. This makes the last line of the while loop very important. It not only changes the value of the condition variable, but it also makes it possible for the test condition to eventually be false. Without this possibility, we will 'lock' the interpreter into performing the same task over and over again... forever. This is known as an infinite loop, which is essentially a loop that never completes. An infinite loop will run either until it crashes the computer or it receives a timeout from a modern interpreter after a certain period of time.

For Loops

In JavaScript there is a second option for creating a loop statement which is more compact and slightly more confusing. For loops can achieve the same thing as a while loop with fewer lines of code, but a code that is a bit unintuitive for the novice programmer. In a for loop, the variable declaration, the condition and the changing value of the variable is all done at the same time. As an example let us write the previous age-growing while loop with a for loop syntax.

```
for (var age = 1; age < 18; age++) {
		document.write("Another year has passed. You are now "+age+" years old.<br>");
}
document.write(age+"? Congratulations! You are now an adult.");
```


First Steps in JavaScript
File Edit View Favorites Tools Help

```
Another year has passed. You are now 1 years old.
Another year has passed. You are now 2 years old.
Another year has passed. You are now 3 years old.
Another year has passed. You are now 4 years old.
Another year has passed. You are now 5 years old.
Another year has passed. You are now 6 years old.
Another year has passed. You are now 7 years old.
Another year has passed. You are now 8 years old.
Another year has passed. You are now 9 years old.
Another year has passed. You are now 10 years old.
Another year has passed. You are now 11 years old.
Another year has passed. You are now 12 years old.
Another year has passed. You are now 13 years old.
Another year has passed. You are now 14 years old.
Another year has passed. You are now 15 years old.
Another year has passed. You are now 16 years old.
Another year has passed. You are now 17 years old.
18? Congratulations! You are now an adult.
```

Image 14. An example of a for loop

As evident from the image, the resulting content is exactly the same in both cases. Each loop begins with the for keyword. This is the followed by a set of parentheses that contains three parts.

1) The first part, which is applied only once at the beginning of the statement, declares the condition variable, age , and sets its initial value to 1. The initial value is used as a starting point for the number of times the loop will repeat and can be any number. If this variable had been initialized earlier in the script, the var keyword would have been unnecessary.

2) The second part is the condition itself, which is evaluated every time before the loop is executed. As long as the condition is true, the for statement will keep repeating the code. When the condition is false, the loop will be instructed to stop running whichin our example is when the age variable becomes greater than or equal to 18.

3) Finally, the third part determines the rate at which the condition variable is changed. In our example the condition variable increases by one after each cycle, but this is not mandatory. The condition variable can get larger or smaller and can increase/decrease by any amount we set.

To finish the structure of the for loop we have curly brackets that enclose the code that we want to use within the loop, which in our example is to write a line of text on the web page with document.write() .

Loops and Arrays

Loop statement are very useful when we want do handle array variables. In JavaScript, we can use loops to cycle through array items, and when necessary, perform an action on each item.

Earlier in this book we created an array, called months , that contained the twelve months of the year.

```
var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
"Sep", "Oct", "Nov", "Dec"];
```

Using either a while or a for loop we will go through each item in the array and print it to the page. We will access the array items by using their index and we will subsequently increase the index value in each loop cycle. If we use a while statement we will have the following code:

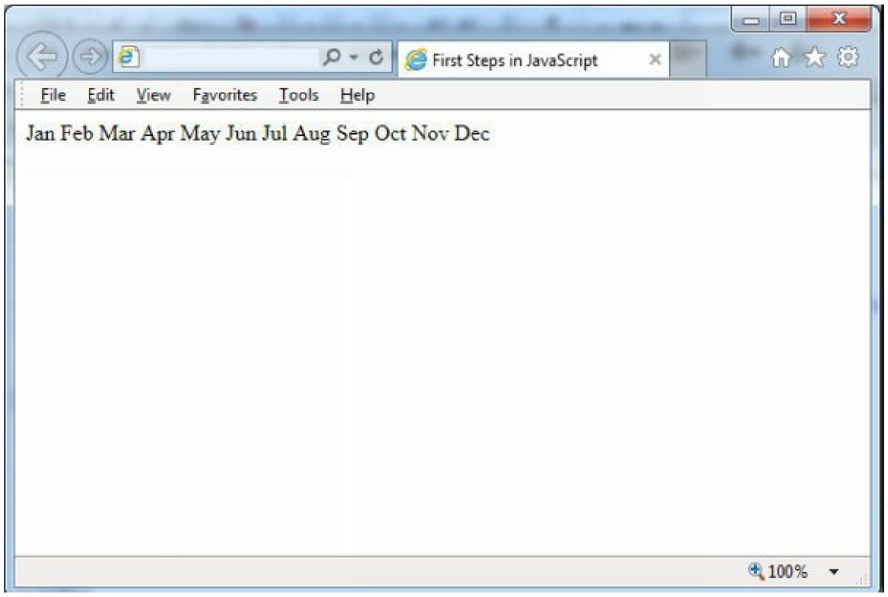
```
var counter = 0;
```

```
while (counter < months.length) {
        document.write(months[counter] + " ");
        counter++;
}
```

We initiate a counter variable that has the same starting value as the index of the first array item, which is 0. Following, we begin a while loopthat runs until the counter value is less than the length of the array. The condition, counter < months.length inquires whether the current value of the counter variable is lower than the number of array items by comparing it to the array's length property. Each time the while loop runs we print the value of a single array item and we subsequently increase the value of the counter variable by 1.

If we want to perform the same action with a for loop we can use the following code:

```
var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
"Sep", "Oct", "Nov", "Dec"];
for (var counter = 0; counter < months.length; counter++) {
        document.write(months[counter] +" ");
}
```



```
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Image 15. Displaying array contents using loops

Chapter 3: Functions

By now we have seen that when writing JavaScript code we are actually writing detailed instructions for what we want to happen one step at a time. In real life, we need detailed instructions only for the first time we perform the action. Afterwards we familiarize ourselves with the steps and do the actions automatically. For example, let us go back to the first time we used a touchscreen smartphone. To turn on the phone we had to perform two unknown actions: click a button to initiate the screen and swipe across the screen to unlock the phone. For each subsequent time we wanted to turn our phone on, we didn't need these instructions; we had already committed them to memory.

JavaScript has a similar mechanism that memorizes steps of a frequently used action, and this mechanism is called a function. A function is a series of steps we create in the beginning of our script in order to use it whenever we need those steps performed. We write the code only once and we run it any time we desire.

This chapter will explain the fundamentals behind using functions. We will learn what a function is, how to define and structure functions, and how to call functions in our scripts.

Creating Functions

The main purpose of a function is to perform a series of actions through a single command. The actual task that the function will perform is dependent on the code itself. It can be something simple as writing a single line of text in the browser, to complex calculations such as evaluation the final price of a shopping basket including discounts and shipping methods. Except for making our scripts more portable, functions are useful because they are reusable. Rather than rewriting blocks of code, we can use the function as many times as necessary. This becomes especially helpful when the functions are lengthy and perform complex tasks.

To define a basic function in JavaScript it is necessary to declare it with its name and code. When choosing a name for a function it would be preferable if we select a name that reflects its purpose. Basically, the name of the function should indicate what the function does.

For our initial example we will use the keyword function followed by the function name writeText and a set of parentheses to declare a function:

```
function writeText() {
            document.write("The answer is 42.");
}
```

By using the keyword function we inform the interpreter that what follows is a series of steps that need to be remembered. The steps are listed between two curly braces, which mark the beginning and the end of the JavaScript code that is a part of the function. In the example above, the steps will be remembered as the name of the function, which is writeText . Hence, whenever we want to initiate this sequence of steps it will be sufficient to only write out the function name followed by empty parentheses. This is also known as calling a function.

```
writeText();
```

In this basic example, every time we call the function within the main script the text *“The answer is 42.”* will be written on the web page. For a more precise and functional example, let us modify a previous script and create a function that will check the age of the user.

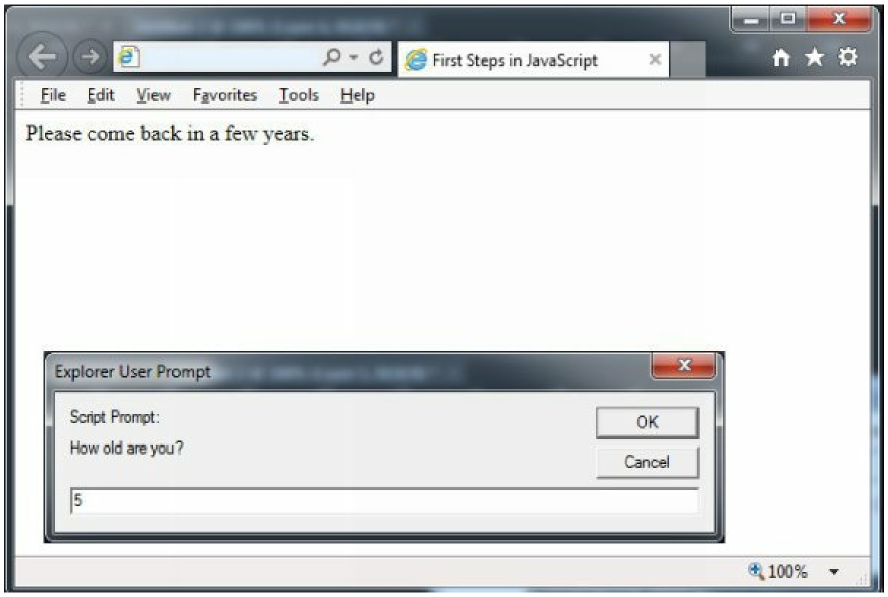
```
function checkAge() {
   if (age >= 18) {
         document.write("You have reached the age of wisdom, please
enter and have some fun.");
   }
   else {
         document.write("Please come back in a few years.");
   }
}
```

The JavaScript code that is placed in a function is stored in the browser's memory, waiting for us to call the function whenever we need to perform that specific action. In this example, the function checks how the age variable relates to the value of 18. Like mentioned previously, to call the function we will simply need to write out the name of the function, followed by a pair of parentheses. We will also add a prompt() which asks the user to provide the value for his/her age.

```
var age = prompt("How old are you?");
checkAge();
```

The complete script would be:

```
function checkAge() {
   if (age >= 18) {
      document.write("You have reached the age of wisdom, please
enter and have some fun.");
   }
   else {
   document.write("Please come back in a few years.");
   }
}
var age = prompt("How old are you?");
checkAge();
```


First Steps in JavaScript
File Edit View Favorites Tools Help
Explorer User Prompt
OK
Cancel
100%

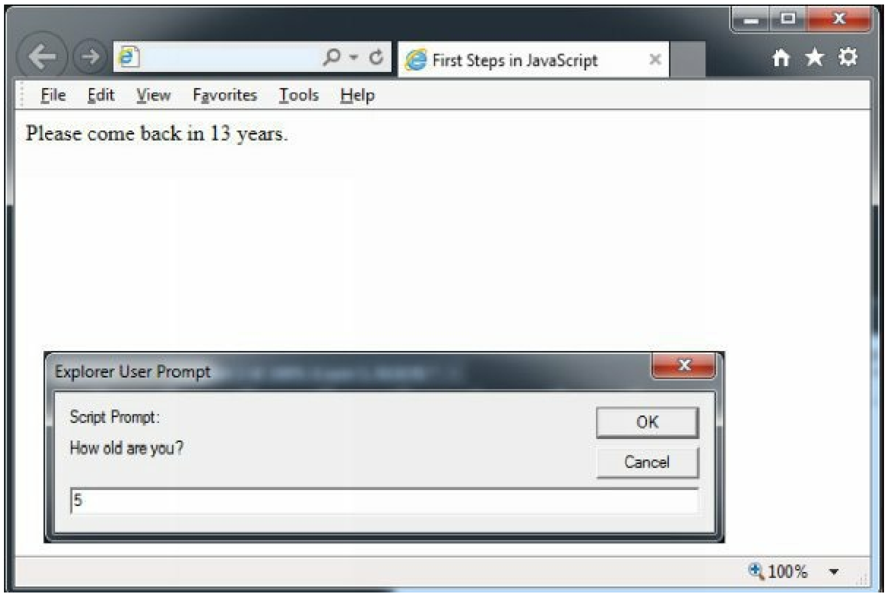
```
Please come back in a few years.
```

```
Script Prompt:
How old are you?
5
```

Image 16. Implementing a function to check the users' age

Additionally, we can use what we learned in previous chapters and make this function more intelligent. Instead of just receiving the answer "Please come back in a few years.", we can calculate the actual difference between 18 and the age entered by the user. To do this we will first introduce a variable named difference in the function, than calculate that difference as 18-age if the entered value for age is less than 18, and finally print the difference variable along with the text.

```
function checkAge() {
            var difference;
            if (age >= 18) {
            document.write("You have reached the age of wisdom,
please enter and have some fun.");
            }
            else {
            difference = 18 - age;
            document.write("Please come back in "+difference+"
years.");
   }
}
var age = prompt("How old are you?","");
checkAge();
```


First Steps in JavaScript
File Edit View Favorites Tools Help
Explorer User Prompt
OK
Cancel
100%

```
Please come back in 13 years.
```

```
Script Prompt:
How old are you?
5
```

Image 17. Smart check age

Sending Information to Functions

We can send one or more values to the function, thus increasing its versatility. When we send these values, the function is capable of processing this information before it comes back with an answer. The values we send are also known as arguments, while the process of sending information to a function is called 'passing an argument'. Once the argument is received the function will use this data while carrying out the sequence of actions as specified in the code.

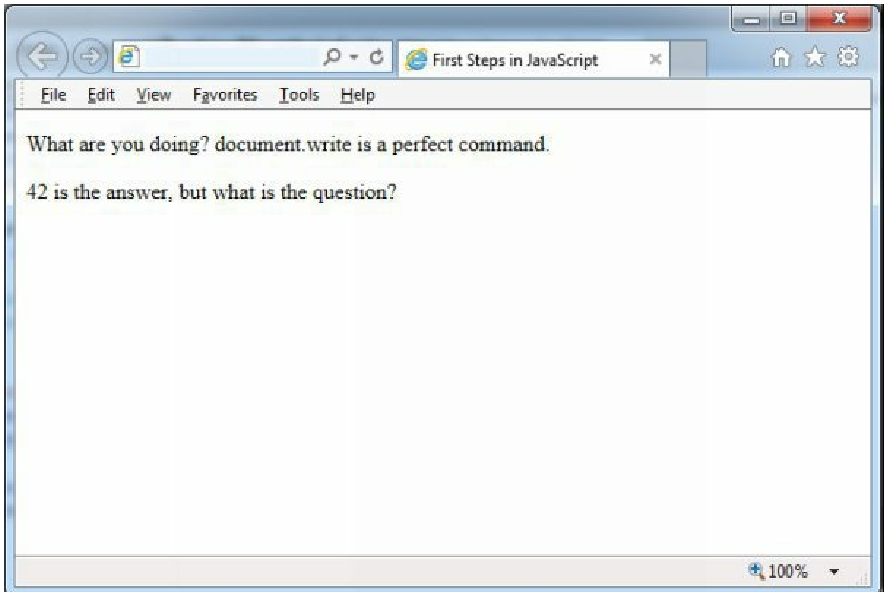
The basic syntax would look like the following:

```
function functionName(argument) {
   //JavaScript code
}
```

Arguments are set on the first line of the function inside the set of parentheses, which we previously left blank. The values brought in as arguments automatically become declared variables within the function. These variables use the names given inside the parentheses and do not need to be further declared with the var keyword.

As the most basic example, let us create a function that will shorten the syntax of the document.write() command.

```
function print(message) {
                document.write(message);
}
print("<p>What are you doing? document.write is a perfect command.
</p>");
print("<p>42 is the answer, but what is the question?</p>");
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

What are you doing? document.write is a perfect command.

42 is the answer, but what is the question?

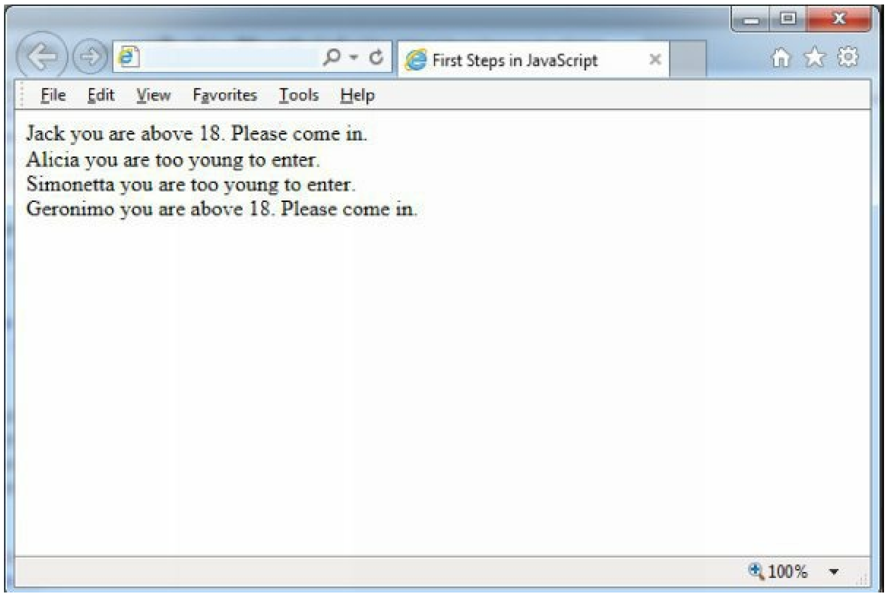
Image 18. Substituting the document.write command with an argument function

This function, named print() , can accept one argument. When the function is called, this argument is passed on to the function and is stored as a variable named message . When the document.write() command is initiated within the function it writes the contents of the message variable on the page.

As mentioned previously, functions are not limited to processing single arguments. We can pass as many arguments to the function as needed as long as each argument is specified in the function and the function is called with the same number of arguments in the same order.

As an example let's revisit the checkAge() function and try to make it more versatile. We will provide the variables name and years to the function and have it respond with a personalized message.

```
function checkAge(name, years) {
if (years < 18) {
        document.write(name+" you are too young to enter.<br>");
}
else {
        document.write(name+" you are above 18. Please come in.
<br>");
}
}
checkAge("Jack", 80);
checkAge("Alicia", 8);
checkAge("Simonetta", 15);
checkAge("Geronimo", 30);
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
Jack you are above 18. Please come in.
Alicia you are too young to enter.
Simonetta you are too young to enter.
Geronimo you are above 18. Please come in.
```

Image 19. Checking the age for multiple user with a single function

In the example above, we initiate the checkAge() function several times, each time with two different values, one for name and another for age. Whenever we call the function, both values are passed to the function as arguments. The first argument is stored as the name variable and the second value is stored as the years variable. Following, the years variable is used in a conditional statement to evaluate whether it is less than or greater than 18. Correspondingly, the name variable is used in the personalized message displayed through the document.write() command. We end each personalized message with a <br> tag in order to have the messages displayed in a separate line.

Note: *You can send any type of JavaScript variable or value to a function: string, number, boolean, array, etc.*

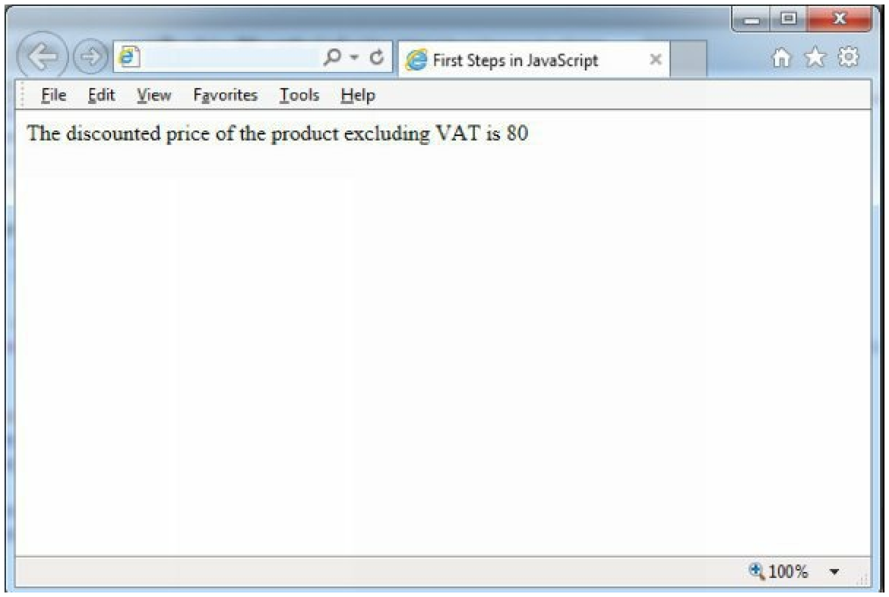
Retrieving Information from Functions

Now that we have seen how to send information we can "push" functions even further and get information back from them. To retrieve a value from the function to the main script we can use the return statement. The generic syntax would look like the following:

```
function functionName(argument1, argument2) {
// JavaScript code
return data;
}
```

For a more specific example let us assume that we want to learn the price of a product after a discount.

```
function checkPrice(price, discount) {
        var total = price - price * discount / 100;
        return total;
}
var actualprice = checkPrice(100, 20);
document.write("The discounted price of the product excluding VAT
is "+actualprice);
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
The discounted price of the product excluding VAT is 80
```

Image 20. Retrieving information from a function

The checkPrice function accepts two values as arguments, the price of the product and the discount that needs to be applied. Once these valued are accepted they are stored in the price and discount variables respectively. The function then calculates the discounted price and assigns this calculated value to the variable total . The mathematical formula price-price*discount/100 is just the formula used to make this calculation. Finally, the value for the variable total is returned to the main script with the return statement.

What is passed down to the script is the value of the total variable, not the variable itself. In order to use this 'returned' value in the script, we need to store it inside a variable. In the above example, we call the checkPrice() function to calculate a 20% discount to a price of $100 and assign the resulting value to the actualprice variable. This variable is then used in the main script and can be displayed on screen with the document.write() command.

We are not required to store the return value in a varible. We can also use it directly in the document.write() command.

```
document.write("The discounted price of the product excluding VAT
is "+checkPrice(100, 20));
```

We have to be aware that as soon as the browser's JavaScript interpreter encounters a return statement, it will exit the function. This means that if there are any lines of code after the return statement they will not be executed. Therefore, the return statement should be the last line of code within the function.

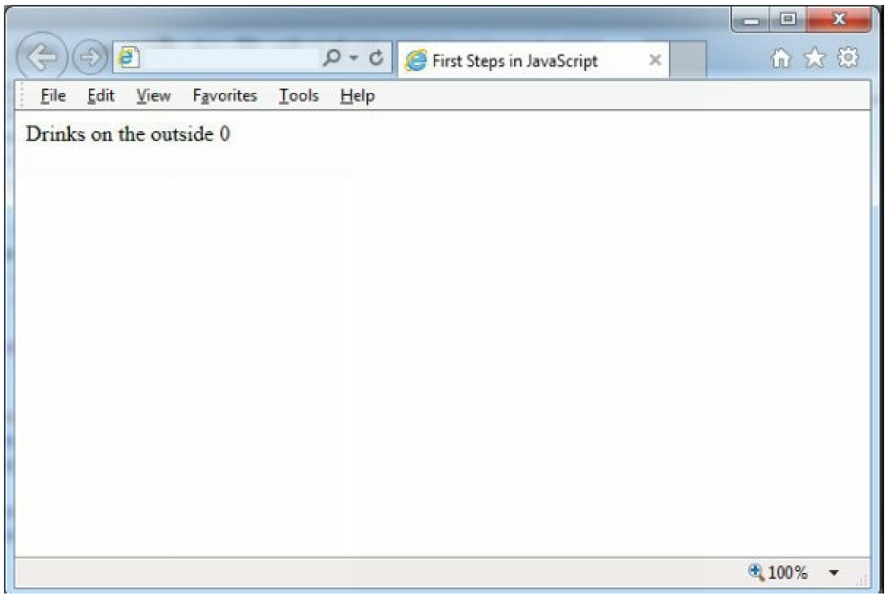
Local and Global Variables

By now we are aware that functions provide us with a great advantage by decreasing the amount of code we have to write. However, before we continue we need to clarify one more aspect which is the behavior of variables inside of a function and outside of the function. There is a difference as to how these variables are treated which is very relevant for our coding practices.

For the JavaScript interpreter, the variables declared inside of a function are treated differently than variables declared outside of a function. The so-called 'scope' of the variables is different in each situation. The variables that exist inside a function are not visible to the rest of the script, they are valid only for the function itself. This means that function variables have a local scope. On the other hand, variables which are declared in the main part of the JavaScript code are meaningful to all parts of the script. This means that all existing functions in a script can access the variables which are created in its main body. These variables have a global scope.

To clarify the difference let us look at a few examples where we will calculate the number of drinks we have during a fun night out.

```
var drinks = 0;
function nightOut() {
        drinks = 5;
        document.write("Drinks in the function ", +drinks);
}
document.write("Drinks on the outside ", +drinks);
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

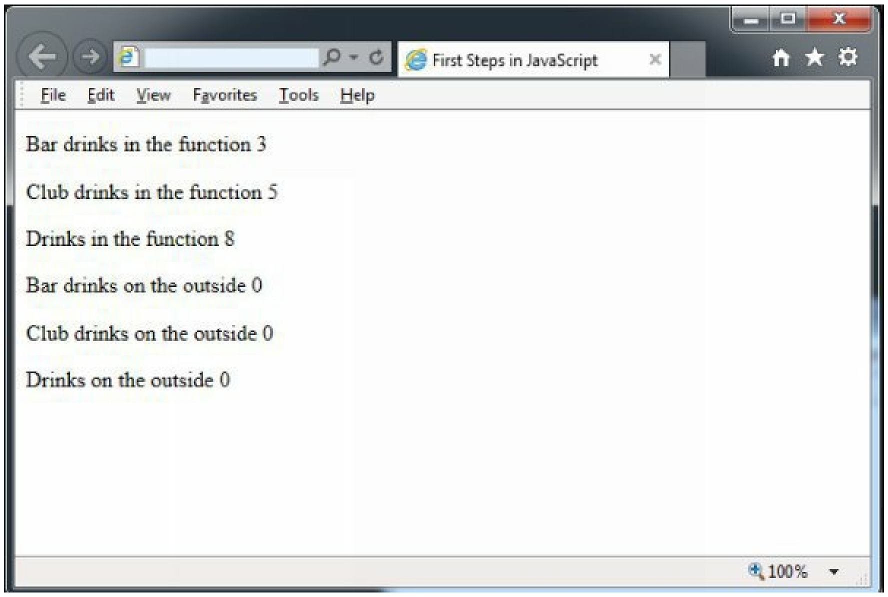
```
Drinks on the outside 0
```

Image 21. Local vs. global variables example

As we begin the day with 0 drinks the variable drinks is declared in the main body of the JavaScript code with the assigned value of 0. As the drinks variable is declared in the main code it has global scope, meaning that it is accessible to all other functions. For this reason, when we display this variable with the document.write statement both in the function and in the main code we get the assigned value of 5.

Now, let us assume that we want to track the number of drinks at each place we visit with the following code:

```
var drinks=0;
var bardrinks = 0;
var clubdrinks = 0;
function nightOut() {
                var bardrinks = 3;
                var clubdrinks = 5;
                var drinks = bardrinks + clubdrinks;
                document.write("<p>Bar drinks in the function ",
+bardrinks);
                document.write("<p>Club drinks in the function ",
+clubdrinks);
                document.write("<p>Drinks in the function ", +drinks);
}
nightOut();
document.write("<p>Bar drinks on the outside ", +bardrinks);
document.write("<p>Club drinks on the outside ", +clubdrinks);
document.write("<p>Drinks on the outside ", +drinks);
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
Bar drinks in the function 3
Club drinks in the function 5
Drinks in the function 8
Bar drinks on the outside 0
Club drinks on the outside 0
Drinks on the outside 0
```

Image 22. Local vs. global variables extended example

Like in the previous example, the variable drinks isdeclared in the main body of the script and therefore it has a global scope. However, in this variation we also declare two additional variables, bardrinks and clubdrinks , within the function. These two variables will have a local scope and can be accessed only in the function itself. Hence, when we display these variables with the document.write() statement within the function we see values, while when we display these variables in the main code we don't see any values.

Chapter 4: Objects

The real world is filled with objects, a random example being cars, TV's, and dolls. Most of these objects are made up of many different parts (ex. cars have doors, a TV's have screens, dolls have heads) and can also do one or more actions (ex. transport people, watch images). The programming world, specifically JavaScript, is also full of objects.

We have reached the point where we need to take our JavaScript knowledge to the next level. For this purpose we will look at the more advanced concepts starting with objects and object-oriented programming. Understanding these concepts will provide us with a vital set of tools that we can even apply when learning other programming languages.

What Are Objects?

To represent physical things and concepts in the programming world we need to use objects. But before we use programming objects let us examine an object from the real world, a car. If we want to describe a car we will talk about its characteristics such as color, make and model. We might expand our description to number of doors, seats, engine type, maximum speed or any other set of characteristics. We can also talk about how we use the car, from turning the ignition key, to driving or listening to the stereo, or even how we use parts of the car, like reclining the seats. We can even compare different car "versions" based on this outline.

In programming, the car would be described as an object. The characteristics of the car such as color, make, model, would be defined as the properties of the car object. The things we can do with the car, like driving or listening to music are represented by what is known as methods. And finally, each actual representation of a car is called an instance. For example, in the following image you are looking at two instances of a car.

Image 23. Two identical cars

Creating Objects

There are two approaches to creating objects in JavaScript, we can either use an object constructor function or literal notation. Let us start with an object constructor example.

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

The first part of this statement is familiar,we use the var keyword to define a car variable. As we want to define the variable as an object we use the new operator followed by the Object() constructor. This will instruct JavaScript to create an empty object and assign it the name car.

Once the empty object has been created we can start defining its properties. Each property will have a name and a value, with each name/value pair describing a particular instance of the object. For our car objectwe can assign the properties color , make and model .

```
var car = new Object();
car.color = "red";
car.make = "Ford";
car.model = "Mustang";
```

We can notice that when defining properties we use the object name, followed by a dot, followed by the property name. This is known as dot syntax, where the dot is what connects the property (or method) to the object.

To define a method for the object, we need to create a function that will become a part of the object. To begin, let us create a simple method that will display a notification that the car has been turned on.

```
car.startCar = function () {
        document.write("The engine has started.<br>");
}
```

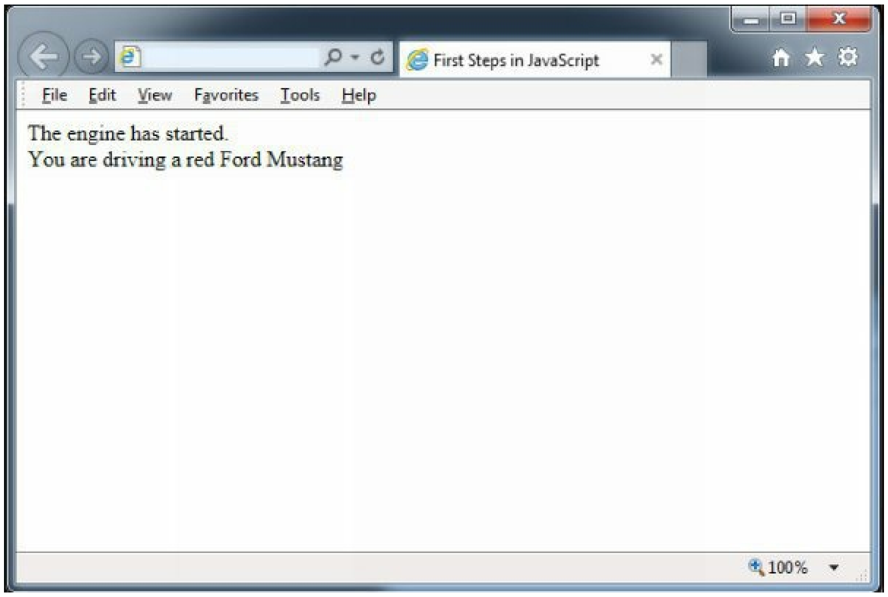
As we are familiar with functions we can see that the startCar method is created with the function keyword, followed by a series of statements placed within curly braces. Since this code only creates the method, in order to see this method in action we have to call it in the script using the object name.

```
car.startCar();
```

The complete code would be:

```
var car=new Object();
car.color = "red";
car.make = "Ford";
car.model= "Mustang";
car.startCar = function () {
        document.write("The engine has started.<br>");
     document.write("You are driving a "+car.color+" "+car.make+"
"+car.model);
}

car.startCar();
```



First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
The engine has started.
You are driving a red Ford Mustang
```

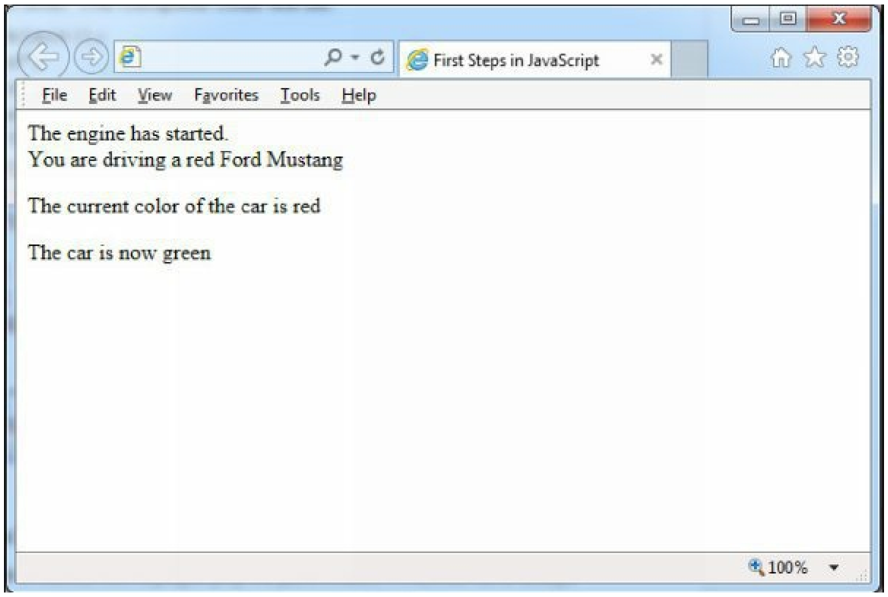
Image 24. Defining a car object

We can use methods for anything that we can use functions for, and that is anything that we can imagine. We can even use a method to change the properties of the object itself. As an example let us create a method that will change the color of the car.

```
car.changeColor = function (othercolor) {
        this.color = othercolor;
        document.write("The car is now "+this.color);
}
```

We can notice two differences from the previous method: we are passing an argument to the method and we are using a new keyword, this . The keyword this is used to refer to the object itself in order to allow us to assign new property values or even creating new properties at the same time. The complete code will be:

```
var car = new Object();
car.color = "red";
car.make = "Ford";
car.model = "Mustang";
car.startCar = function () {
        document.write("The engine has started.<br>");
        document.write("You are driving a "+car.color+"
"+car.make+" "+car.model);
}
car.changeColor = function (othercolor) {
        this.color = othercolor;
        document.write("<p>The car is now "+this.color);
}
car.startCar();
document.write("<p>The current color of the car is "+car.color+ "
<br>");
car.changeColor("green");
```


First Steps in JavaScript
File Edit View Favorites Tools Help
100%

```
The engine has started.
You are driving a red Ford Mustang

The current color of the car is red

The car is now green
```

Image 25. Changing the properties of an object

> ***Note:*** *We can always distinguish a method from a property by the parentheses. A method name always ends in parentheses.*

Another way to create objects in JavaScript is with object literal notation. With this approach we use curly brackets to enclose the properties and methods for the object. To reproduce the car object with literal notation we would write the following code:

```
var car = {
color: "red",
make: "Ford",
model: "Mustang",
startCar: function () {
        document.write("The engine has started. <br>");
        },
changeColor:  function (othercolor) {
        this.color = othercolor;
        document.write("The car is now "+this.color);
        }
}
```

From the example it is noticeable that in literal notation each name/value pair is separated by a colon, including the method function. We can have as many name/value pairs as necessary as long as each pair is separated with a comma.

Unknowingly, we have been using JavaScript objects in our statements. For example, our most frequently used statement in all our examples is the document.write() statement. As a matter of fact in using this statement we have actually been addressing the document object and telling it to execute the write() method. We will learn more about the document object in the next chapter

Chapter 5: Document Object Model (DOM)

In the previous chapter we learned how objects work and we even looked at some of JavaScript's predefined objects. We are now prepared to investigate the most important web object that is the document object. The document object will primarily help us to gather information about the web page.

The document object is an object that is created by the browser for each new HTML page. When it is created, JavaScript allow us access to a number of properties and methods of this object that can affect the document in various ways, such as managing or changing information. As a matter of fact we have been continuously using a method of this object, document.write() , in order to display content in a web page. Nevertheless, before exploring properties and methods we will first take a look at the Document Object Model (DOM).

Fundamental DOM Concepts

We are aware that when the web browser receives an HTML file it displays it as a web page on the screen with all of the accompanying files like images and CSS styles. Nevertheless, the browser also creates a model of that web document based on its HTML structure. This means that all the tags, their attributes and the order in which they appear is remembered by the browser. This representation is called the Document Object Model (DOM) and it is used to provide information to JavaScript how to communicate with the web page elements. Additionally, the DOM provides tools which can be used to navigate or modify the HTML code.

The Document Object Model is a standard defined by the World Wide Web Consortium (W3C) that is used by most browser developers. To better understand the DOM, let us first take a look at a very simple web page:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Party Schedule</title>
<style type="text/css">
.current {
```

```
                color:red;
        }
        .finished {
                color:green;
        }
        </style>
        </head>
        <body>
        <h1 id="partytitle">Party Plan</h1>
        <ul id="partyplan">
          <li id="phase1">20:00 - Home warm-up</li>
          <li id="phase2">22:00 - Joe's Bar</li>
          <li id="phase3">00:00 - Nightclub 54</li>
        </ul>
        </body>
        </html>
```

On a web page, tags wrap around other tags. The <html> tag wraps around the <head> and <body> tags. The <head> tag wraps around tags such as <title> , <meta> and <script> . The <body> wraps around all content tags such as <p> , <h1> through <h6> , <ul> , <table> and so on.

This relationship between tags can be represented with a tree structure where the <html> tag acts as the root of the tree, while other tags represent different tree branch structures dependent on the tag hierarchy within the document. In addition to tags, a web browser also memorizes the attributes of the tag as well as the textual content within the tag. In the DOM each of these items, tags, attributes and text, are treated as individual units which are called nodes.

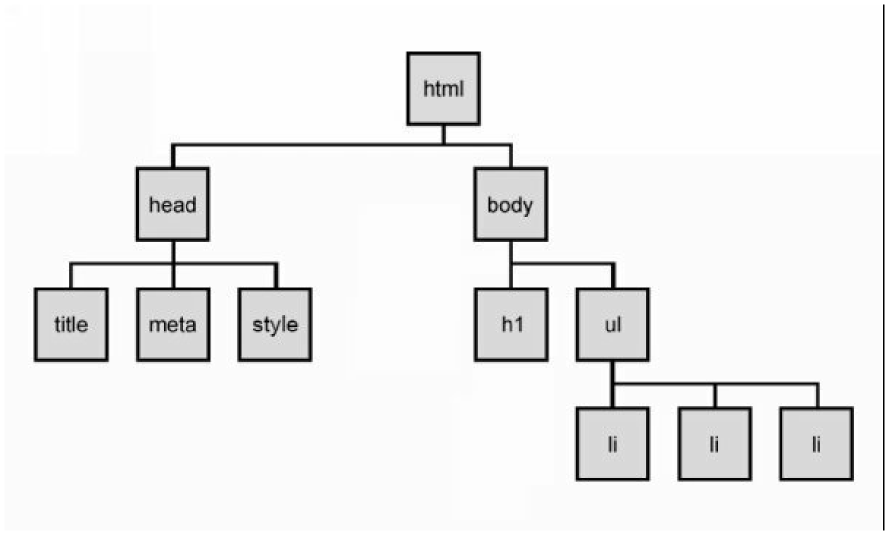


Image 26. Tree structure of an HTML document

In the tree structure for our basic HTML page the <html> element acts as a root element, while the <head> and <body> elements are nodes. In defining this relationship we can also refer to <html> as the parent node, and the <head> and <body> elements as child notes. In turn, both the <head> and <body> elements contain child nodes and so on. When we reach an item that contains no other child node we terminate the tree structure at that node, also known as a leaf node.

Selecting Document Elements

With the DOM structure in place, JavaScript can access the elements within the document in several different ways, dependent on whether we want to select individual or multiple elements. In all approaches we first have to locate the node representing the element we need to access and subsequently use the content, child elements and attributes of that node.

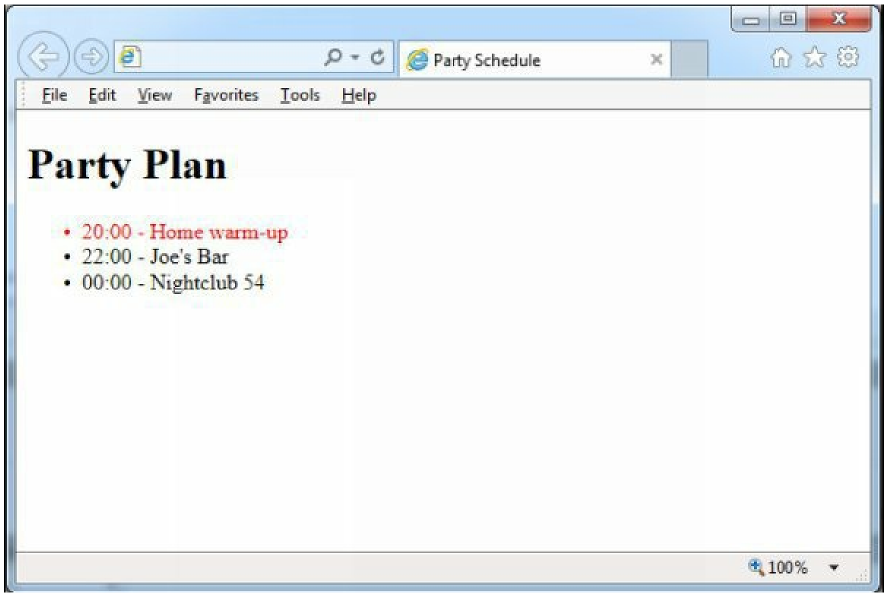
Selecting Individual Elements

To select individual elements we most commonly use the getElementById() method. This method will let us select an element with a particular ID attribute applied to its HTML tag. This method is the most efficient way to access an element if we follow the presumption that the ID attribute is unique for every element within the page. In the following example we will access the element whose ID attribute has the value 'phase1' :

```
var firststop = document.getElementById("phase1");
```

By using the getElementById() method on the document object means that we are searching for the element with this ID anywhere on the page. Once the 'phase1' element is assessed, which in our case is the first <h1> element, the reference to this node is stored in the firststop variable and we can use JavaScript to make changes. As an example we will assign the attribute class with the value 'current' to this element. We will include this code in a <script> tag in the <head> section of our document.

```
var firststop = document.getElementById("phase1");
firststop.className = "current";
```



Party Plan

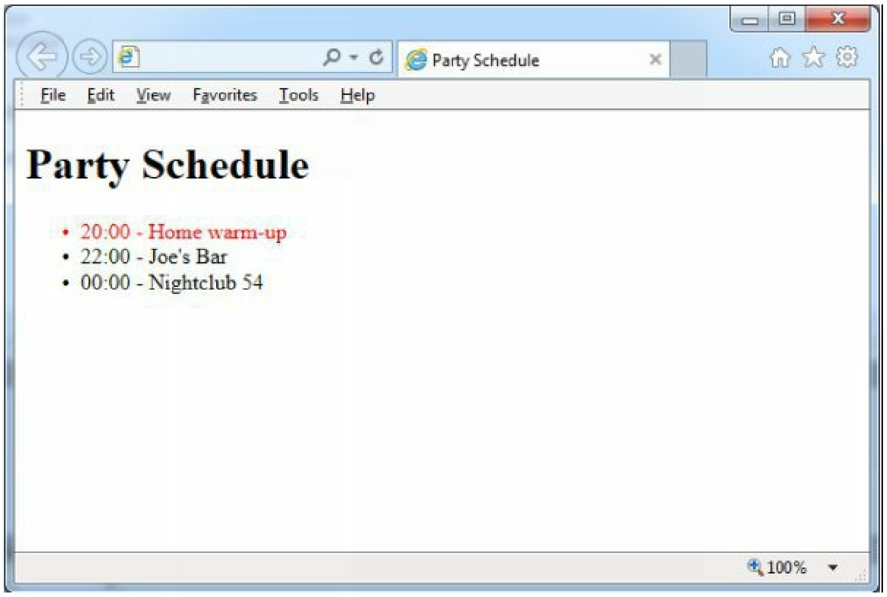
- 20:00 - Home warm-up
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54

Image 27. Changing the style of a page element

> ***Note:*** *In some browsers we have to either put the <script> tag before the closing </body> tag or in an external .jsfile in order for the code in this chapter to work .*

If we want to collect the text from a node, we can use the textContent property. More importantly, we can also use the textContent property to change the content of the node. In the following example we will first select the element that has the value 'partytitle' in its id attribute and assign it to the title variable. Then we will effectively change the text of this element by changing the textContent property of the title variable. Let us add the following lines to our JavaScript code:

```
var title = document.getElementById("partytitle");
title.textContent = "Party Schedule";
```



Party Schedule

- 20:00 - Home warm-up
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54

Image 28. Changing the content of a page element

Selecting Group Elements

While sometimes selecting an individual element will be sufficient, other times we may need to select a group of elements. For example, we might need to select all <li> tags on a page, or all elements that share a class attribute. In these cases JavaScript offers the following two methods:

- **getElementsByTagName()** – a method which will let us select every instance of a particular tag.
- **getElementsByClassName( )** – a method that retrieves all elements that share a particular class name.

Selecting a group of elements means that the method will return more than one node. This collection of nodes is known as a NodeList and will be stored in an array-like item. Each node will be given an index number, starting with 0, while the order of the nodes will be the same order in which they appear on the page. Although NodeLists look like arrays and behave like arrays, semantically they are a type of object called a collection. As an object, a collection has its own properties and methods which are rather useful when dealing with a NodeList.

The following example will select all <li> elements and assign their node references to the schedule variable.

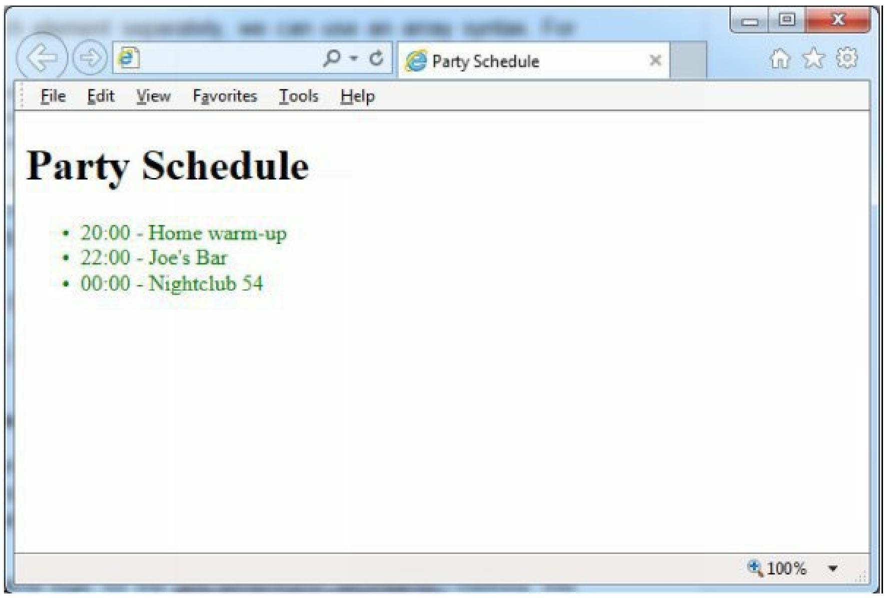
```
var schedule = document.getElementsByTagName("li");
```

If we want to access each element separately, we can use an array syntax. For example:

```
var item1 = schedule[0];
var item2 = schedule[1];
var item3 = schedule[2];
```

However, when we select a group of items we usually want to interact with the whole group. As an example, let us assign the class attribute with the "finished" value to all <li> elements. For this purpose we can use a loop to go through each element in the NodeList.

```
var schedule = document.getElementsByTagName("li");
for (var i = 0; i < schedule.length; i++) {
        schedule[i].className = "finished";
}
```



Party Schedule

- 20:00 - Home warm-up
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54

Image 29. Changing the class attribute for all <li> elements

Similarly to working with arrays, when working with collections we can use the length property to determine the size of the collection. We can then use this information in a for loop in order to effectively go through every NodeListitem and assign the "finished" class attribute.

We can use exactly the same logic for the getElementsByClassName() method. We will get a NodeListstored in a collection with each node having an index number. Like with the getElementByTagName() method, we can access individual items and manage the collection through its object properties and methods.

Traveling Through the DOM

When we use any of the previously discussed methods to select an element node, we can also select other elements in relation to this elements. This type of relative selection is considered as an element property.

previousSibling & nextSibling

The previousSibling and nextSibling properties refer to adjacent elements on the same DOM level. For example, if we select the second <li> element with the id value "phase2" , the "phase1" <li> element would be considered a previousSibling , while the "phase3" <li> element would be nextSibling . In the case where there is no sibling, (ex. the "phase1" element has no previousSibling), the value of this property remains null.

In the following example we select the <li> element which has "phase2" as a value for its id attribute and we change the class attribute for both the selected element and its previous sibling.

```
var secondstop = document.getElementById("phase2");
var prevstop=secondstop.previousSibling;
secondstop.className = "current";
prevstop.className = "finished";
```

Parents & Children

We can also travel to different levels of the DOM hierarchy using the selected element as a starting point. If we want to move one level up we can use the parentNode property. For example, if we have the second <li> element selected we can refer to its parent element, the <ul> element, with the following syntax:

```
var secondstop = document.getElementById("phase2");
var upperelement = secondstop.parentNode;
```

Alternatively, if we want to move one level down, we can use either the firstChild or the lastChild property. In the following example we have selected the <ul> element with "partyplan" as a value for its id attribute. Using the firstChild property we refer to the first <li> element of this list, while with the lastChild property we refer to the last <li> element of this list.

```
var plan = document.getElementById("partyplan");
var child1 = plan.firstChild;
var child2 = plan.lastChild;
```

Adding and Managing Content

Until this point we discussed how to find elements in the DOM. The more interesting aspect are the approaches to managing content within the DOM.

Changing HTML

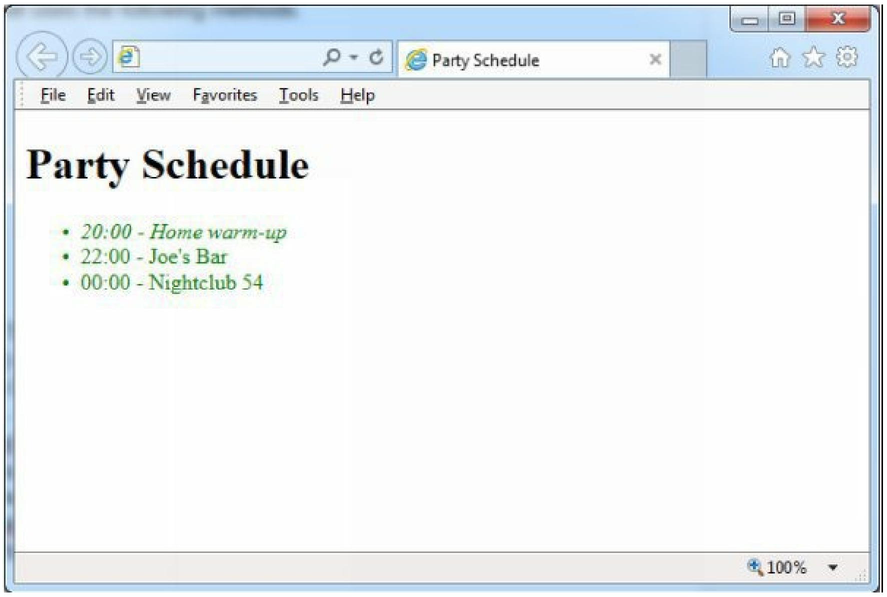
We already talked about the textContent property, but this property retrieves only text values and ignores the subsequent HTML structure. If we want to edit the page HTML we have to use the innerHTML property. This property can be used on any element node and it is capable of both retrieving and editing content.

```
var liContent = document.getElementById("phase1").innerHTML;
```

When retrieving the HTML from the <li> element with "phase1" as a value for its id attribute, innerHTML captures the whole content of the element, text and markup, as a string variable. If we apply the same syntax for the <ul> element, the innerHTML property will capture all of the <li> items.

We can also use the innerHTML property to change the content of the element. If this content contains additional markup, these new elements will be processed and added to the DOM tree. For example, let us add the <em> tag to the first <li> item in the party list:

```
var firstStop = document.getElementById("phase1");
firstStop.innerHTML = "<em>20:00 - Home warm-up</em>";
```



Party Schedule

- *20:00 - Home warm-up*
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54

Image 30. Adding an <em> element with content to the first list item

DOM Manipulation

A more direct technique to managing document content is to use DOM manipulation. This is a 3-step process that uses the following methods:

1. **createElement()**- The process begins by creating a new element node with the createElement() method. This element node is stored in a variable and it is not yet a part of the DOM.
2. **createTextNode()**- The process continues by creating a new text node with the createTextNode() method. Like in the previous step, this text node is stored in a variable and it is not a part of the document.
3. **appendChild()**- The final step is adding the created element to the DOM tree with the appendChild() method. The element will be added as a child to an existing element. The same method can be used to add the text node to the element node.

As an example let us create a new element that we will add to the existing party list. We will use the createElement() method and add this element to the newPlan variable.

```
var newPlan = document.createElement("li");
```

Following, we will create a new text node and add its content as a value to the newPlanText variable.

```
var newPlanText = document.createTextNode("04:00 - Back to home");
```

We can now assign the content of the text node to the newPlan element by using the appendChild() method.

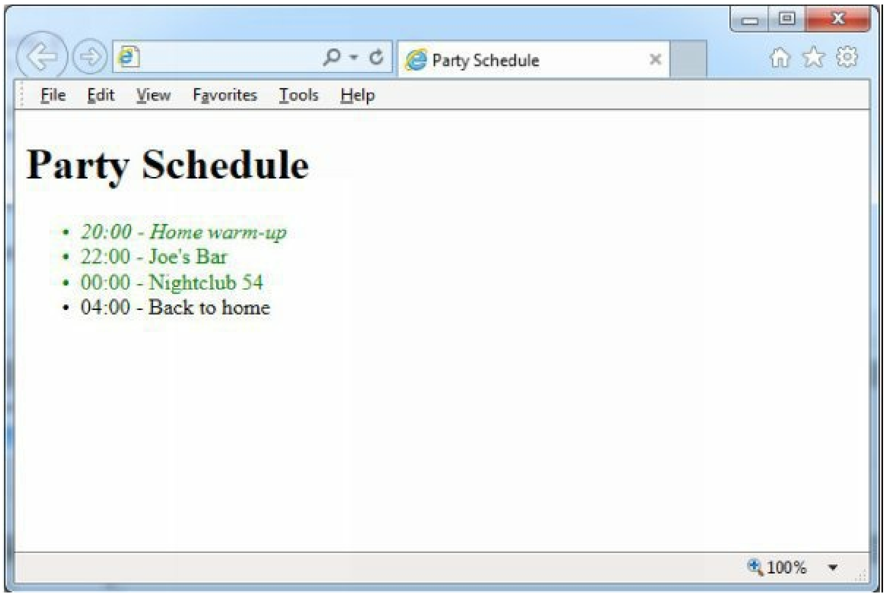
```
newPlan.appendChild(newPlanText);
```

Finally, we would like to add this element to the list. We will use the getElementById() method to select the list through its "partyplan" id , and apply the appendChild() method to attach the newPlan element to the list.

```
document.getElementById("partyplan").appendChild(newPlan);
```

The complete syntax is as follows:

```
var newPlan = document.createElement("li");
var newPlanText = document.createTextNode("04:00 - Back to home");
newPlan.appendChild(newPlanText);
document.getElementById("partyplan").appendChild(newPlan);
```



Party Schedule

- 20:00 - Home warm-up
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54
- 04:00 - Back to home

Image 31. Adding a new <li> element

Using a similar process we can also use DOM manipulation to remove an element from the page. As an example let us remove the <h1> element which acts as the main page heading. We will first select the element through its id attribute with "partytitle" as its value and store that element node in a variable.

```
var removeHeading = document.getElementById("partytitle");
```

Next, we will need to find the parent element whichacts as a container for the <h1> element, which in this case is the <body> element. We can either select this element directly, or use the parentNode property of the previously selected element. In either case we will need to store the parent element in another variable.

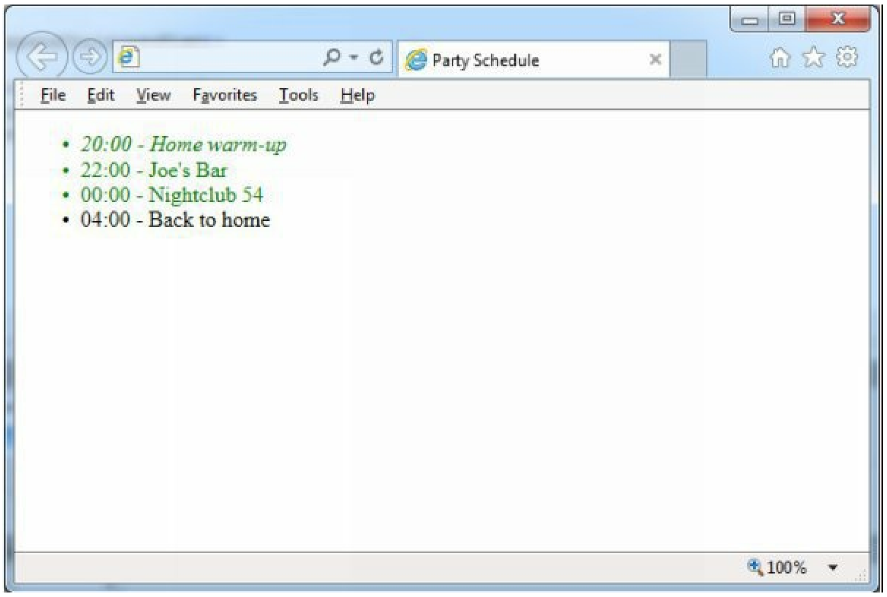
```
var containerForHeading = removeHeading.parendNode;
```

Finally, we will use the removeChild() method on the parent element in order to discard the element that we want removed from the page.

```
containerForHeading.removeChild(removeHeading);
```

The complete syntax is as follows:

```
var removeHeading = document.getElementById("partytitle");
var containerForHeading = removeHeading.parentNode;
containerForHeading.removeChild(removeHeading);
```



- *20:00 - Home warm-up*
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54
- 04:00 - Back to home

Image 32. Removed heading

Chapter 6: Events

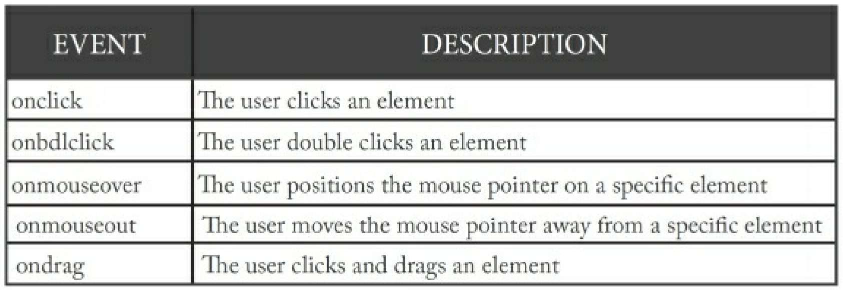
One of the most important concepts in JavaScript is the use of events. An event is simply something that happens, like pressing a key on the keyboard or clicking somewhere on the page. As long as something happens and it has been expected we can respond to it with a specific action. We can intercept the anticipated event and respond in kind by calling a function when that event occurs. As we will see in this final chapter, to make JavaScript anticipate events we will use event handlers or event listeners. But first, let us look at the type of events that we can handle or listen for.

Event Types

When we interact with web pages we initiate a lot of events. We click a link, hover over an element, type text, open the browser, close a tab, copy/paste text, fill-out forms and so on. Almost everything we do triggers an event, and when an event is triggered we usually want to code a function that will react specifically to that event.

There are many types of events that we can react to, but we will only focus on mouse events, keyboard events and browser/object events. The following three tables give a brief description of the most commonly used events for each event type.

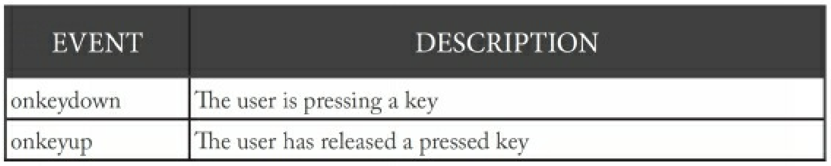
Mouse events occur when the user does something with the mouse like moving, clicking, dragging, etc.



EVENT	DESCRIPTION
onclick	The user clicks an element
onbdlclick	The user double clicks an element
onmouseover	The user positions the mouse pointer on a specific element
onmouseout	The user moves the mouse pointer away from a specific element
ondrag	The user clicks and drags an element

Table 2. Mouse events

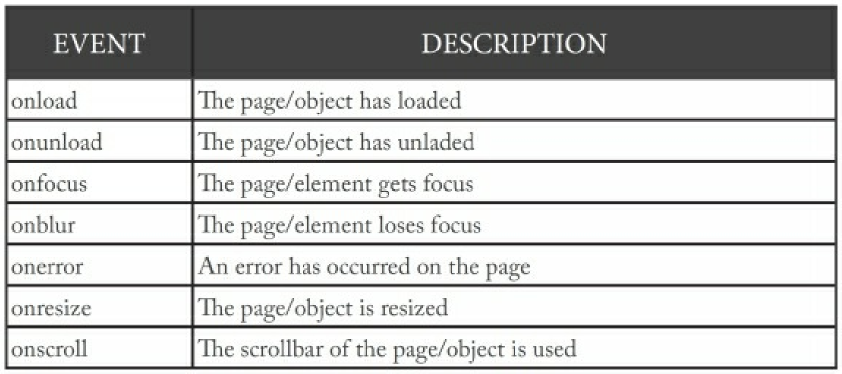
Keyboard events are simpler as they occur either when a key is pressed or depressed.



EVENT	DESCRIPTION
onkeydown	The user is pressing a key
onkeyup	The user has released a pressed key

Table 3. Keyboard events

Browser/object events are more generic events that occur at different times like when the web page is loaded.



EVENT	DESCRIPTION
onload	The page/object has loaded
onunload	The page/object has unladed
onfocus	The page/element gets focus
onblur	The page/element loses focus
onerror	An error has occurred on the page
onresize	The page/object is resized
onscroll	The scrollbar of the page/object is used

Table 4. Browser/Object events

Mouse events are the most common user-based events that occur on a web page. However, with the advent of touch-based devices there is an increased consideration for touch events.

> ***Note:*** *The full list of possible events is rather large and beyond the scope of this book. Please visit the W3C Schools web site for a complete list (http://www.w3schools.com/jsref/dom_obj_event.asp).*

Reacting to Events

In order to program JavaScript to react to an event we have to bind that event to an element on the page via event handlers. Event handlers let us indicate the event we are listening for on any specific element. For this purpose there are three different approaches to event handlers:

- **HTML event handlers** are a set of attributes that can respond to events on the element they are added. Although still present in older web pages, this approach is no longer used as it integrates JavaScript into HTML when it should separate.

- **DOM event handlers** are the approach introduced in the original DOM specification. They are separate from the HTML document and have a strong support in all major browsers. Their main drawback is the limitation of attaching only a single function to an event.

- **DOM event listeners** are the favored approach to handling events, introduced in an update of the DOM specification. They allow one event to trigger multiple functions, however they are not supported by IE8 and earlier versions of that browser.

When dealing with events we first have to select the element that will be the object of interaction. If for example this element is a link, then we need to specify the DOM node for that link element. After we have the element selected we will need to indicate the event that will act as a trigger. In programmer speak this is called "binding the event to the element node". Finally, we have to state the code, which is usually a function that we want to have triggered when the event happens.

In the remainder of the chapter we will focus on event handlers and event listeners and ignore the discarded HTML approach. To follow the examples in this chapter we will use a similar HTML structure like in the Party Schedule example from the previous chapter. The only difference is the link elements that have been added to every list item.

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Party Schedule</title>
<style type="text/css">
.current {
        color:red;
}
```

```
.finished {
        color:green;
}
</style>
</head>
<body>
<h1 id="partytitle">Party Plan</h1>
<ul id="partyplan">
  <li id="phase1">20:00 - <a href="#" id="testLink">Home warm-
up</a></li>
  <li id="phase2">22:00 - Joe's Bar</li>
  <li id="phase3">00:00 - Nightclub 54</li>
</ul>
</body>
</html>
```

Event Handlers

There are many very similar approaches to using traditional DOM event handlers. Some are very compact, while others have more lines of code, but might be more understandable. As an example, let us write a brief script that will change the color of the link once it is clicked. To begin we will first create the function that we want to have triggered when the event occurs.

```
function linkClick(){
        this.className = "current";
}
```

By using the this.className property we instruct the object calling on this function to set its className property to "current" . Now that we have our function in place let us call on the object that will trigger the event by using the getElementById() method and assign its element node to a variable.

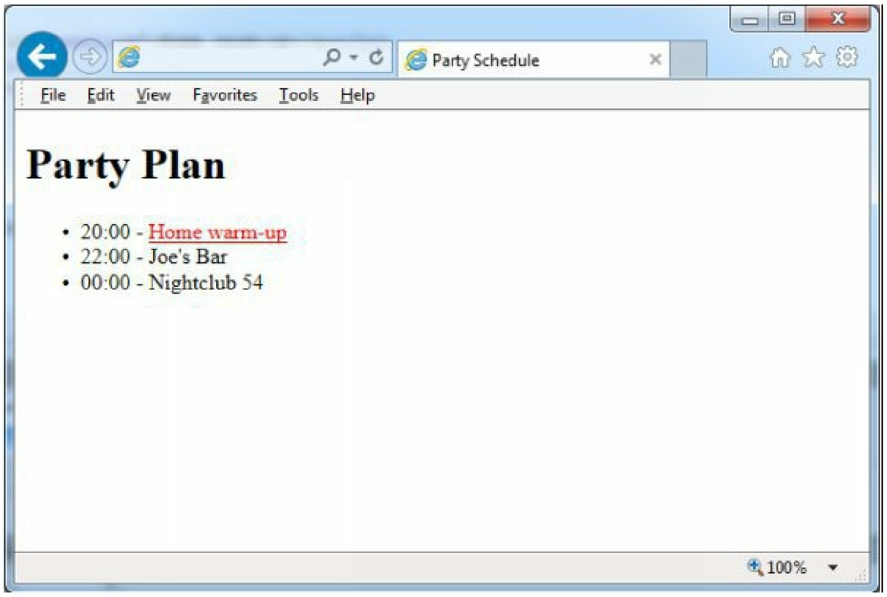
```
var firstLink = document.getElementById("testLink");
```

As we have retrieved the reference to the element node, we can finally construct a proper event handler.

```
firstLink.onclick = linkClick;
```

The complete syntax is as follows:

```
function linkClick(){
                this.className = "current";
}
var firstLink = document.getElementById("testLink");
firstLink.onclick = linkClick;
```



Party Plan

- 20:00 - Home warm-up
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54

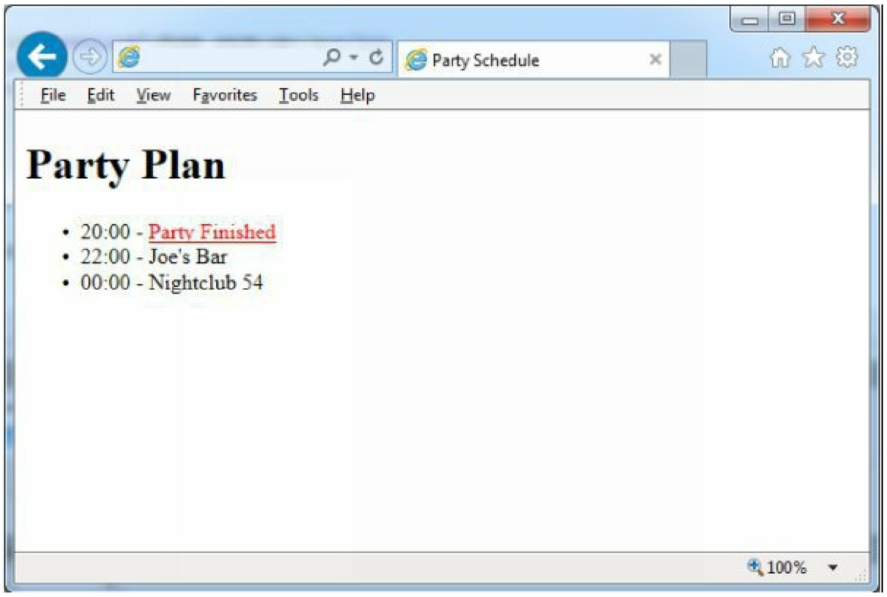
Image 33. Using an event handler to change the link color

Considering that JavaScript is a very flexible language we can actually compact this in two lines of code, but we will lose some of the flexibility.

```
document.getElementById("testLink").onclick = function
linkClick() {
   this.className = "current";
}
```

The linkClick function that is triggered by the onclick event doesn’t have to be simple. We can change more than one property and we can even write complex scripts that deal with different situations. For example, the following code will change the text of the link in addition to the addition of a class.

```
function linkClick(){
                this.className = "current";
                this.textContent = "Party Finished"
}
var firstLink = document.getElementById("testLink");
firstLink.onclick = linkClick;
```



Party Plan

- 20:00 - Party Finished
- 22:00 - Joe's Bar
- 00:00 - Nightclub 54

Image 34. Changing multiple properties on a click event

We can see that in our examples no parentheses are added after the function name when the function is assigned to the event. The reason for this is that when if use the linkClick() syntax with parentheses we would be executing the function and assigning its value to onclick . When we don't use parentheses we assign the function (not the value) to the onclick property. We no longer control how the event handler function is executed as the browser will execute the function for us by automatically passing an Event object to the handler function.

Event Listeners

The more recent innovation to handling events is called an event listener. An event listener deals with multiple functions at the same time, although older browser support is lacking. The event listener is attached to the element via the addEventListener() method which takes three properties.

As an example let us rewrite the previous code and change the event handler to an event listener. The function will remain the same.

```
function linkClick(){
        this.className = "current";
        this.textContent = "Party Finished"
}
var firstLink = document.getElementById("testLink");
firstLink.addEventListener("click", linkClick, false);
```

We can see that instead of the event property, what is attached to the firstlink variable is the addEventListener() method. Within the parentheses we first have the event (without the preceding 'on'), followed by the name of the function and something called event flow. Event flow indicates the order in which the event is captured and it is usually set to false.

Conclusion

When we decide to learn something as complicated as JavaScript, there is a long process ahead of us. It is not sufficient to get acquainted with information, we have to apply that information in practical situations and real-life examples.

In this book we introduced the fundamental concepts of JavaScript and its approach to programming. We learned that JavaScript is a scripting language that enables us to enhance web pages by providing dynamic and interactive content. We looked into the process the browser follows when interpreting web pages as it parses the code element by element and acts on JavaScript instructions. We saw that JavaScript code is embedded into the web page itself; its presence marked with the <script> elements.

Starting from the basic syntax and statement structure, in the second chapter we built up our knowledge with JavaScript's data types and variables. Particularly, we now know that JavaScript supports a number of types of data, such as numbers, text, and booleans. Numbers behave like numbers should, text is represented by strings of characters and is surrounded by quotation marks and booleans are either true or false. These data types are stored in variables, making their values committed to memory so that they can be used later in our code. However, before giving value to a variable we are now aware that we first must declare its existence to the JavaScript interpreter. We also familiarized ourselves with arrays as a special type of variable that can hold more than one piece of data which is managed by means of a unique index number.

In the third chapter we continued to learn the core of the JavaScript language and its syntax. Specifically, we looked into decision making with conditional statements and repetitive code use with loops. We started by understanding if , else and else if statements as the ability to make decisions is essentially what gives the code its "intelligence". Determining whether a condition is true or false can help us decide on a course of action to follow and in using the statements we can choose the block of code that will be executed respectively. Following, we looped the code with while and for loops as it is often necessary to repeat a block of code a number of times. We learned that looping requires initialization, condition testing, and incrementation in order to successfully execute blocks of code a specified number of times.

We finished looking at the JavaScript core scripting capabilities in the fourth chapter by observing functions as reusable bits of code. Although there are main built-in functions available, JavaScript enables us to define and use your own functions using the function keyword. We saw that functions can have zero or more parameters passed to them and can return a value if that is their intended purpose. In relation to functions, we learned about variable scope, where variables declared outside a function are available globally while variables defined inside a function are private to that function and can't be accessed by the main code.

In chapter five we moved on to the vital concept of objects. We learned that JavaScript is an object-based language as it represents things, such as strings, dates, and arrays using this object concept. We created new objects with the object constructor function or literal notation. We also set the properties of those objects and we defined their methods. We also learned how to access these properties and call these methods when needed.

We examined the document object model in chapter six where we saw how the DOM offers means to access web page elements. We learned that the DOM represents the HTML document as a tree structure. We then learned how this tree structure makes it possible to navigate through its "branches" to different elements to use their properties and methods. We saw how we can select individual elements and groups of elements by locating element nodes. We then manipulated the content by either adding new elements or changing the existing structure.

We now have the basic knowledge and understanding to move on to learning more advanced concepts. There are many directions we can take as JavaScript offers a vast amount of directions we can take. We can focus on form interaction and see how we can use JavaScript to evaluate and interact with form data. To do this we also need to learn about serialization to translate the structure and information of the object. We can then personalize the user experience by learning how to store information on local computers with cookies.

We can push things even further with AJAX, creating uninterrupted applications that don't require page refreshing for server communication. We can explore JavaScript frameworks such as jQuery or Modernizr to see how we can take JavaScript to its limit and easily create sophisticated high-class applications. Finally, we can learn how to create the perfect code with advanced error handling and using sophisticated debugging tools.

Glossary

Argument - a variable used by a function that has been passed to that function.

Array - a collection of values in a single data type.

Boolean - a data type accepting true/false variables.

Constructor - a way to create a new instance of an object.

CSS – Cascading StyleSheets

DOM - a collection of definitions that allow a program written in JavaScript to interact with the objects on the web page.

ECMAScript - the core specification of the JavaScript language.

Event - something that happens on a web page.

Function - a group of statements that have been collected together and given a name.

HTML – HyperText Markup Language

Instance - a specific representation of an object.

JavaScript - a scripting language allowing for improved interaction between users and web pages.

Method - a function attached to a particular object.

Node - a unit representation from the DOM tree structure.

Object - an individual item.

Operator - a way to change/evaluate the content of a variable.

Parsing - the process of reading the source code of the program in order to determine what the code is supposed to do.

Properties - characteristics of a particular object.

String - a data type for text elements.Variable – a named location used for storing values.

W3C – World Wide Web Consortium

About ClydeBank Technology

We are a multi-media publishing company that provides reliable, high-quality and easily accessible information to a global customer base. Developed out of the need for beginner-friendly content that is accessible across multiple formats, we deliver reliable, up-to-date, high-quality information through our multiple product offerings.

Through our strategic partnerships with some of the world’s largest retailers, we are able to simplify the learning process for customers around the world, providing them with an authoritative source of information for the subjects that matter to them. Our end-user focused philosophy puts the satisfaction of our customers at the forefront of our mission. We are committed to creating multi-media products that allow our customers to learn what they want, when they want and how they want.

The Technology division of ClydeBank Media is composed of contributors who are experts in their given disciplines. Contributors originate from diverse areas of the world to guarantee the presented information fosters a global perspective.

Contributors have multiple years of experience in IT systems, networking, programming, web development and design, database development and management, graphic design and many other areas of discipline.

For more information, please visit us at www.clydebankmedia.com or contact info@clydebankmedia.com