Modern Javascript

Introduction

How to Learn using O'Reilly School of Technology Courses

Note Because you are taking the JavaScript course, we assume that you already know HTML. If you need a refresher, feel free to email your mentor at **learn@oreillyschool.com** for information about the O'Relly School of Technology (OST) HTML course.

To learn a new skill or technology, you must sit down and experiment with the technology. The more experimentation you do, the more you learn. Our learning system is designed to maximize your experimentation and help you **learn how to learn** the technology. Here are some tips for learning:

- **Learn in your own voice**--Work through your own ideas and listen to yourself and take ownership of your new skill. We avoid lengthy video or audio streaming and keep spurious animated demonstrations to a minimum, because we know you'll learn more and better by practicing.
- **Take your time**--Learning takes time -- rushing through won't help you learn. By taking your time to try new things, you'll discover more than you (or we) ever imagined.
- **Create your own examples and demonstrations**--In order for you to understand a complex problem, you must understand its simpler parts. For this reason, you'll build each demonstration piece by piece yourself, following our guidance.
- **Experiment with your ideas and questions**--Wander from the path to see what's possible. We can't possibly anticipate all of your questions and ideas, so you must experiment and create.
- **Accept guidance, but don't depend on it**--Try to overcome difficulties on your own. Going from misunderstanding to understanding on your own is the best way to learn. It's how you LEARN HOW TO LEARN. Our goal is to make you independent of us. Of course, our instructors are here to help in extreme cases.
- **Do REAL projects**--Real and live projects, as opposed to simulated ones, are more rewarding and challenging. They help to ensure that you learn what's involved in real-world situations. Our Learning Sandbox is built around a real development tool for doing real jobs. We'll give you projects to do in the objectives and quizzes for each lesson.

Understanding the OST Learning and Work Space Environment

Since this may be your first OST course, it's important for you to be introduced to the **Learning Sandbox** learning environment. Below is the **CodeRunner® Editor**, which will serve as your workspace for experimenting with code we give you, as well as your own ideas. CodeRunner is a multi-purpose editor that allows you to create applications in many technologies. One of the technologies that CodeRunner will enable you to create in is HTML.

All of the communication tools and learning content goes in the upper part of the screen, as you can see. Frequently we will ask you to type code into the CodeRunner Editor below and experiment by making changes. When we want you to type code on your own, you'll see a white box like the one below with code in it (in this case it is not necessary to type the suggestion):

Type the following into CodeRunner below:

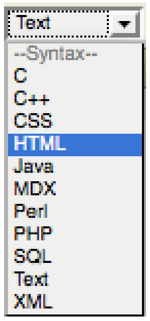
```
 We'll ask you to type code that
    occurs in white boxes like this into CodeRunner below. Every time you see a white b
ox like this,
    it's your cue to experiment.
```

Similarly, when we want you to "observe" some code or result, we will put it in a gray box like this:

Observe the following:

```
Gray boxes like this will be used for observing code. You
    are not expected to type anything from these "example" boxes.
```

Since CodeRunner is a multi-purpose editor, you'll need to use the correct **Syntax**. In this course you will only be using **HTML** Syntax. The Syntax menu looks like this:



Throughout this course we will give you more tips for using the learning environment as needed.

Using the CodeRunner® Editor

This is an OST Learning Lab, where you learn by trying the suggested lessons and experimenting on your own. In these labs, you'll script for the web and make your very own webpages using the Editor below. Let's take a closer look.

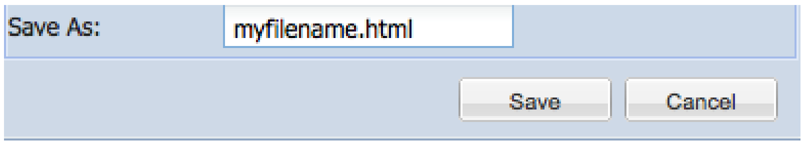
In the bottom half of the window, you should see the **CodeRunner® Editor**. In the space provided, please type "HELLO WORLD." Be sure that **HTML** is selected from the drop-down menu.

Type the following into the Editor below:

```
HELLO WORLD
```

Saving and Retrieving Your Page

Let's practice saving your page on the OST server. Go ahead and click on the button. Choose a name for your file and type it in the Save As text box. You also need to include the html extension (.html) or the browser won't know to treat your code as html.



You should also practice using the Save as button to save another version of your file.

Pretty easy, eh? After successfully saving your file, anybody can go on the web, type the URL (http://yourusername.oreillystudent.com/myfilename.html) in the location bar of their browser, and see this page.

To retrieve your page click simply click on the Load File icon or double-click the file name in the File Browser window at the left.

Previewing Your File

After you type "HELLO WORLD", click on the Preview icon, which looks like this:

The first thing that happens when you Preview a file is that CodeRunner checks to see whether this file has been saved previously. If it has, the page will be saved with the same name. If not, the Save File window will open.

Note Keep in mind that every time you Preview a file, your changes will be saved. Think about whether you want the previous code overwritten or not. If not, use File Save As before you Preview.

Once the file has been saved, another browser window or tab should open and show you your first web page.

You're Ready to Go!

At this point you should be comfortable using the Preview button and with saving and retrieving your files. You're going to use this a lot, so if you feel like it would help, go practice it a couple more times.

Throughout the rest of these learning labs, whenever I want you to type something into your document, I'll put it into a white box. When I just want you to look so I can show you something, I'll put it into a gray box. You'll probably want to save your page after each lesson so you can go back and get it later, but that's up to you.

Introduction

About Javascript

JavaScript, as a language, is more complex than HTML. While HTML is a markup language, JavaScript is a programming language. If you're not familiar with programming languages, JavaScript might be a bit confusing at first. There are many "words" in this language. If you try to memorize each "word" you see throughout the lessons, you may feel overwhelmed. Instead, concentrate on learning things like:

- objects, methods, and properties, and how to refer to them.
- when to utilize event handlers.
- the syntax for writing functions.

Armed with this knowledge, you'll be able to understand the JavaScript language, even if you don't necessarily know every "word." This is the same philosophy we used in the HTML class; we didn't cover every available tag, but instead empowered students with the knowledge of how tags work and the right tools to be comfortable learning new ones as needed.

Note All of the examples, quizzes, and objectives for this course should be completed in the Sandbox. If you have never taken an OST Programming course, please click here to learn about the different features of this tool.

JavaScript is not Java

Okay, so you've decided to learn JavaScript. Great! At this point, you probably don't know too much you about it though, at least beyond its general usefulness for web programming. Before we dive into Javascript though, let's go over the differences between JavaScript and Java.

JavaScript and Java are both widely used in web development. Because their names are so similar, you might assume that they are one and the same, but that's not the case.

Java

You may be familiar with Java Applets (programs written in Java that can only be run in a browser) on web pages, but applets represent only a small portion of the capabilities of the Java language. Java was originally developed to be used as a programming language for things like word processors, calculators, car computers, watches, PDAs (Personal Digital Assistants), and microwaves. One of the goals the creators of Java had was to create a language that could be used by multiple platforms, like those I just mentioned. In the process of developing this language, programmers found that Java was great for the internet as well, and worked with a wide variety of platforms found there (UNIX, Windows, Macintosh, and so on).

Java is based on C++, the language currently used most by software developers. The creators of Java incorporated the best qualities of C++, while getting rid of the worst (like pointers...ugh!).

Java is an **Object-Oriented (OO)** language, which means that you work with objects of some type and apply actions to them. This is in contrast to procedural programming where you apply objects to some procedure. (If you have a programming background, you're probably aware that my definition is pretty condensed, but it will help us to contrast Java with JavaScript later.)

Java is also a **compiled** language, which means that the program is written in text, then converted to a computer program (which you can't read) when it's executed.

JavaScript

JavaScript has nothing to do with Java. JavaScript, originally created by Netscape, is ported to Internet Explorer as **JScript**. It is a language devoted to enhancing HTML. Because it can be used to improve design, create interactive webpages, and create cookies, it is the most popular scripting language on the internet.

JavaScript is an **Object-Based** language; it uses many of the concepts of OO programming, but it's not completely Object-Oriented.

JavaScript gives you (the programmer) more control over your webpage. Java simply uses the document as an environment from which to run applications.

JavaScript is an **interpreted** language, meaning it is written and executed as is. Unlike Java, no special program is needed to convert JavaScript before the browser can understand it.

Looking Ahead...

So, we've touched on the basics about the origins and goals of JavaScript. By the time you've finished this course, you'll be able to apply the powerful abilities of JavaScript to your own documents. What kind of stuff will you be able to do? Well, just take a look at this example below. After working through the course, I'm confident you'll be able to create something like this and a whole lot more:

- OST Tic Tac Toe

Recycling JavaScripts

Using Javascript Scripts

You don't need to know anything about JavaScript in order to start using it on your page. You can recycle scripts written by other people.

Here's a case scenario: you're surfing the internet and you come across a web page that does something interesting using JavaScript. You think to yourself, *that would look great on my web page!* You're sure you can replicate the HTML tags that are working with the script, but you don't know anything about JavaScript. No problem! To recycle scripts from other web pages, you need to know just these **four** things:

Number 1: The Script Itself

First things first -- you need to know where to get the script. JavaScripts are always placed within **script tags** and may be placed within **comment tags** as well. Here's a typical script:

OBSERVE:

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

<!--

function displayTime(){

   var date = new Date()
   var hours = date.getHours()
   var min = date.getMinutes()
   var sec = date.getSeconds()

   document.getElementById("myText").value=hours+":"+min+":"+sec

   setTimeout("displayTime()",500)
}

//-->

</script>
```

In the above example, the **script tags are red** and the **comment tags are blue**.

The code between these tags composes the script itself. The scripts are often located in the head of the HTML document, but they can be just about anywhere. The code could seem like a foreign language to you right now. Don't worry! We'll discuss it in detail later. For now, I just want you to recognize that this is the script. You would need to copy all of this code into your HTML document in order to recycle it. It's a good idea to place it in the same section you found it. In other words, if it was in the head on the original page you're borrowing from, put it in the head of your page as well.

If you see a web page that's using JavaScript and you want to recycle it, move your mouse over the page and click the **right** mouse button. A list will appear. Select **View Source** from the options in the list to see the code used to generate the page. Then try to **locate the script**, then copy and paste it into your document.

Let's give that a try. Locate a script and copy it into your HTML document. Click here!

Number 2: The Event

Now we'll adress the component being used to make the script go! It's called an **event handler**. You don't need to know too much about event handlers in order to recycle them. (We'll go over event handlers in greater detail in a future lesson.)

Event handlers are **always** located within HTML tags, usually within **body**, **input**, or **anchor** tags. Event handlers specify when and why the browser should execute a script.

Even if you aren't familiar with specific event handlers, there is a reliable way to locate them. First, find the function (you'll learn about these later) that contains the script you want to use:

OBSERVE:

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

<!--

function displayTime(){

   var date = new Date()
   var hours = date.getHours()
   var min = date.getMinutes()
   var sec = date.getSeconds()

   document.getElementById("myText").value=hours+":"+min+":"+sec

   setTimeout("displayTime()",500)
}

//-->

</script>
```

The name of the **function** is **displayTime**. We are able to identify it because the word **displayTime** follows the word **function**.

Once you find the function name, check out the body of the HTML document. Inside the tags, you'll find something that looks like this:

onevent="functionName()"

In this example, the event handler is located within the body tag. It looks like this:

OBSERVE:

```
<body onload="displayTime()">
```

Another helpful way to locate event handlers is to be aware that they begin with the word "**on**."

Click here to see the page you looked at before. **View Source** on the page and see if you can locate the event handler that starts the script. Copy and paste it into your HTML document (in the same place that corresponds to where you found it on the original document).

Number 3: The Related Objects

The script that you are recycling displays the time in a text field, so you will have to create a text field in your HTML document where the time can be displayed as well. You must reproduce the object **exactly** as it appears in the document from which you are recycling the script. The code to create this particular text box looks like this:

OBSERVE:

```
<form>
   <input id="myText" />
</form>
```

Failure to reproduce the code exactly as it appears originally, could cause the script to run incorrectly or not at all.

See if you can recycle this script. Be sure to copy all necessary components!

Number 4: Copyright Infringement

So now that you have this information at your fingertips, you can expolit all of those who came before you by recycling their Javascript scripts and using them yourself, right? WRONG! You should never take a script without asking. Sure, it's likely that no one would ever know, let alone complain, but it's always a good idea to ask the script owner for permission and avoid any complications.

Often, information regarding allowable usage, persons to contact in order to request permission, or other copyright information is included in the script itself and is commented out.

Customizing the Script

So now you can recycle a script "as-is" from someone else's web page. It's also possible to customize scripts to suit your own needs. See if you can figure out how to use the scripts in this lesson to display your own personal message instead of the messages currently displayed.

Surf the web a bit and see if you can find some JavaScript to recycle and customize!

Event Handlers

Event Handlers

In this lesson you'll create an interactive web page that will allow vistors to your page to perform an action (or **event**) that will cause something to happen on the screen. Move your cursor over the image below to see an example of that:

Events allow you to create web pages that will respond to input by visitors and are part of what makes JavaScript so fun and powerful!

Copy and paste the script below into the Editor. We'll be using this script throughout the rest of this lesson. Don't panic if you don't understand the script—you're not supposed to yet!

Note Keep in mind that every time you **Preview** an html file, your changes will be saved. If you don't want your previous code overwritten, think about using **Save As** before you **Preview**. You can use the **Go Back** icon to retrieve previous versions even after your file has been saved, if necessary.

In HTML mode, type the code below into the Editor:

CODE TO TYPE:

```
<html>
<head>
<script type="text/javascript">
<!--

function rollon(){
document.getElementById("myImage").src="http://courses.oreillyschool.com/javascript1/im
ages/zap.gif"
}

function rolloff(){
document.getElementById("myImage").src="http://courses.oreillyschool.com/javascript1/im
ages/pow.gif"
}

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

</html>
```

Okay, now that you have the script, let's get back to our discussion of events. Events make things happen. We're going to add some HTML to the script. The HTML we add will include **event handlers** to allow interaction between the HTML and the script.

Note For the rest of this lesson, be sure that the script remains unchanged. This includes everything between the <script> and </script> tags. Any changes you make will be applied to the body of your document.

In HTML mode, type the following code just below the closing head tag in yor Editor:

CODE TO TYPE:

```
<body>
<img src="http://courses.oreillyschool.com/javascript1/images/pow.gif" id="myImage" onm
ouseover="rollon()" />
</body>
```

Click Preview. When your document comes up, move your mouse over the image that is displayed. Pay close attention to what happens. Does the image change? If not, go back and try it again.

When you move your mouse over the image, an event is triggered, causing the image to change. Take a closer look at the code:

OBSERVE:

```
<body>
<img src="http://courses.oreillyschool.com/javascript1/images/pow.gif" id="myImage" onm
ouseover="rollon()" />
</body>
```

When you move the **mouse over** the **image**, the **rollon** function is executed.

onmouseover is the **event handler**. Notice that **onmouseover** is used as an attribute of the **image tag**, just like the **src** attribute. The event handler is set equal to the action that should take place when the event occurs. In this case, when the mouse is moved over the image, the **rollon** function is executed. This function is defined within the script tags. You'll learn how to create your own functions later in the course. Although there are many types of event handlers, they're all used within HTML tags. *Certain* event handlers that can be used within *certain* HTML tags. At the end of this lesson, there's a table that lists all of the possible combinations.

Now that you know a little something about event handlers, we'll have some fun! In HTML mode, replace the body of the document with this code:

CODE TO TYPE:

```
<body>
<img src="http://courses.oreillyschool.com/javascript1/images/pow.gif" id="myImage" onm
ouseover="rollon()" onmouseout="rolloff()" />
</body>
```

Click Preview. Move your mouse over and off of the image this time. Pay close attention to what happens.

When you mouse over the image, the same thing happens as in the previous example. When you move your mouse off of the image though, it changes back to the original image. This is done using a new event handler called **onmouseout**. onmouseout is also used within the image tag; it's triggered when the mouse leaves the image.

Now that you're familiar with events used in the image tag, let's talk about a special type of event. Most events begin with **on**, but there is one that doesn't. This event can only be used within the **anchor** tag.

Whenever you select a link on a web page, you're actually triggering an event. The event causes the web page specified with the href attribute to appear. Instead of setting the href attribute equal to a URL, try using it to call a script. In HTML, replace the body of the document with this code:

CODE TO TYPE:

```
<body>
<a href="javascript:rollon()">
<img src="http://courses.oreillyschool.com/javascript1/images/pow.gif" id="myImage" />
</a>
</body>
```

Click Preview. When the page comes up, **click** the image and see what happens.

You should have seen the image change. Let's analyze the syntax:

OBSERVE:

```
<body>
<a href="javascript:rollon()">
<img src="http://courses.oreillyschool.com/javascript1/images/pow.gif" id="myImage" />
</a>
</body>
```

When you click on the **image**, the **JavaScript** function called **rollon** is executed.

The only difference between this anchor tag and a "normal" anchor tag is that the **href** attribute is set equal to a **JavaScript** function. The specific function is listed after the colon (**:**), which allows you to execute JavaScript using the natural clicking event of the anchor tag. In other words, the event you wish to execute is triggered when the link is clicked. All of the commands necessary to execute the desired command are located within the link rather than the URL.

Let's see events in action with a different HTML tag. In HTML, replace the body of the document with this code:

CODE TO TYPE:

```
<body>
<img src="http://courses.oreillyschool.com/javascript1/images/pow.gif" id="myImage" />
<form>
<input type="button" value='Click Me' onclick="rollon()" />
</form>
</body>
```

Click Preview. This time you should see the image change when the button is clicked.

onclick is an event that can be used with a **button**. When you click the button, the event is triggered, causing the rollon function to be called.

In HTML, type the following into the Editor below:

```

<body>
<img src="http://courses.oreillyschool.com/javascript1/images/pow.gif" id="myImage" />
<form>
<input type="text" size="20" onFocus="alert('You focused the text field!'); this.blur()
" />
</form>
</body>
```

Click Preview. You'll see a text field. When you click in the text field, it is brought into focus, thereby triggering the onFocus event. The result is that an alert message appears.

What do you suppose **this.blur();** does? Blur is the opposite of focus. Focus means to bring an object into view. Blur means to take an object out of view. When you take the text field out of focus, you get stuck in a loop of alert messages. Take **this.blur** out of your code and click PREVIEW.

You've learned a lot about events in this lesson! You can click here for a list of events and the types of HTML objects that you can use with them.

See you in the next lesson!

Introduction to Scripting

Intro to Scripting

By now you know quite a bit about JavaScript. You know about the origins of JavaScript, the basics of recycling JavaScripts for you own use, and how to use event handlers to call those scripts. Now you're ready to begin writing your own scripts!

Your First Script

Let's start with a basic script that will write text to a web page. This type of script is referred to as **static** because it always does the same thing. It requires no event handlers; instead, it's executed when the web page is loaded by the browser.

In HTML mode, type this code into your editor:

CODE TO TYPE:

```
<html>
<body>

</body>
</html>
```

Click Preview. You'll see a blank web page. We'll use JavaScript to write text to this blank web page. In HTML, type this cde into the Editor:

CODE TO TYPE:

```
<html>
<body>

<script type="text/javascript">
<!--
window.document.write("Here is some text")
//-->
</script>

</body>
</html>
```

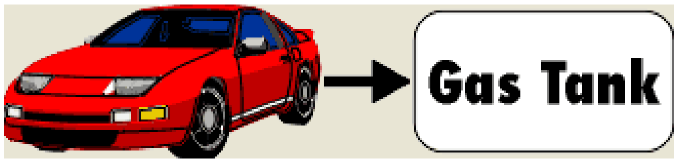
Click Preview. You should see a web page displaying the text **Here is some text**. Congratulations, you have just written your first script!

When including a script within an HTML page, you **must** enclose your scripts within **script tags**. These tags give the location of your script to the browser. The **type="text/javascript"** attribute indicates that the text within the script tags should be interpreted as JavaScript.

Your script should also be enclosed within **comment tags**. These tags hide the script from older browsers that don't support JavaScript. Although most browsers do support JavaScript, it's still a good idea to get into the habit of including them. These tags can also be used to include comments about your code, which can help you to edit your code at a later date.

The line of code between the script and comment tags is a JavaScript command. When the command is executed, the phrase **Here is some text** will be **written** to the HTML **document**.

Notice that there's a period (**.**) between **window** and **document**. It lets the script recognize relationships between **objects**. In **object-oriented programming**, you work on **objects**. In order to work on those objects, you have to define exactly which object you are working on, using hierarchical relationships. Take a look at this diagram:



car.gastank

If you consider a car in pieces, a gas tank is a part of a car. In programming terms, the **car** is the parent of the **gastank** object, and the **gastank** is the child of the **car** object. To reference the gas tank in JavaScript, you would use:

car.gastank

The code instructs the browser to first locate the **car** object, then locate the **gastank** object within the **car** object. So, to reference the car engine using JavaScript, you might use **car.engine**.

In the script that we wrote, we used:

window.document

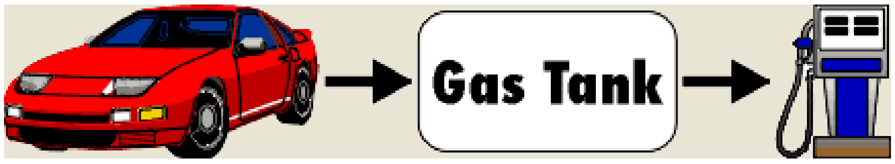
The **window** object represents the browser window. A window object is created automatically with every instance of a <body> tag. The **window** object is the parent of the **document** object. The **document** object represents the entire HTML document and can be used to access all elements in a page.

An Introduction to Methods

Methods define functions performed by an object. Thus, **window.document.write()** calls the **write()** method of the **document** object.

You can think of **objects** as nouns - like **car** and **gastank** or **window** and **document**. You can think of **methods** as verbs - like **fill** or **write**.

Let's say that our car is out of gas, and we need to fill it up:



car.gastank.fill(gas)

fill() is a method of **gastank**. It doesn't make sense to fill the *car* with gas - that would be really dangerous and hard to clean! What you actually do is take your car to the gas station, walk around to the gas tank, and fill up your tank with **gas**. To reference the **fill()** method in JavaScript, you would use:

car.gastank.fill(gas)

This is known as **calling** the **fill()** method.

We use the same concept in the script that we're writing:

window.document.write("Here is some text")

The **write** method is used to write a string of text to the document. It's a method of the document object, not the window object. What do you think would happen if you removed document from the command? In HTML, type this code into the Editor:

CODE TO TYPE:

```
<html>
<body>

<script type="text/javascript">
<!--
window.write("Here is some text")
//-->
</script>

</body>
</html>
```

Click Preview. The script no longer works, and you likely got an error such as, **Error: window.write is not a function** or **Object doesn't support this property or method**. Be sure to go back and put **document** back into your reference!

All methods are followed by a set of parentheses. If a method requires additional information it is included within the parentheses. The **write()** method requires one piece of information -- the text you wish to write to the document. That text is located within the parentheses and is placed inside quotation marks:

window.document.write("Here is some text")

The **Here's some text** par of the code is also known as a **string**. A string is a sequence of characters such as letters, numbers, and punctuation marks. Other methods require different data types; we will learn more about that later.

The above statement does **not** end with a semicolon. With traditional programming languages, like C++ and Java, each code statement must end with a semicolon. Some programmers continue this habit when writing JavaScript, but in general, semicolons are **optional**. Semicolons are required if you want to put more than one statement on a single line though.

Get into the habit of using a reference when you are writing your scripts. That way you can see which objects and methods are available to you, and the type of data the methods require. See if you can use this link to determine which other methods can be used with the document object.

You're doing great so far! Keep it up. More about methods in the next lesson...

Methods and Variables

Methods and Variables

Methods

In the last lesson you wrote your first script and learned about methods. In this lesson you'll learn even more about methods by creating an interactive script. This script will prompt you for input, then it will use the **write()** method to print what you've entered to the document. In HTML mode, type this code into your Editor:

CODE TO TYPE:

```
<html>
<body>
<script type="text/javascript">
<!--

window.document.write(window.prompt("Enter your name:",""))

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

</html>
```

Note If you are using IE7, you may need to change a browser setting for this script to work properly. Check that Tools > Internet Options > Security > "allow websites to prompt for information using scripted windows" is enabled.

Click Preview.

The **prompt()** method returns a value of the text entered into the prompt dialog box. The **write()** method is then used to print the value returned to document.

Expanding on this, you could use the write() method to write additional text, part of which includes the value returned by the **prompt()** method. Let's try that. In HTML mode, type this code into the Editor:

CODE TO TYPE:

```
<html>

<body>
<script type="text/javascript">
<!--

window.document.write("Hi " + window.prompt("Enter your name:","") + ". Welcome
to my page.")

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

</html>
```

Click Preview.

This time your name was added to other text, and then the full text was written to the document. The text located within quotation marks is treated as a constant value, that is, the script uses it as-is. The **+ operator** is used to combine the value returned from the **prompt** method to the text that is used as is. The text that is combined and the final text that is written to the document are known as **strings**.

Note You'll learn more about operators in a later lesson.

Variables

You may have noticed in the last example that the command line got kind of confusing because methods were used within methods: strings were added to one another within a method. You could do that in one step like we did above, but the more you program, the more you'll experience that it's well worth it to break your scripts down into smaller chunks. Not only does it make your initial programming easier, but it also makes it easier to find any mistakes when your browser gives you an error message.

A major tool at your disposal for simplifying scripts is **variables**.

Let's revisit our car example. Recall that when we wanted to **fill** the **gastank** on our **car** with **gas**, we used the javascript equivalent:

car.gastank.fill(gas)

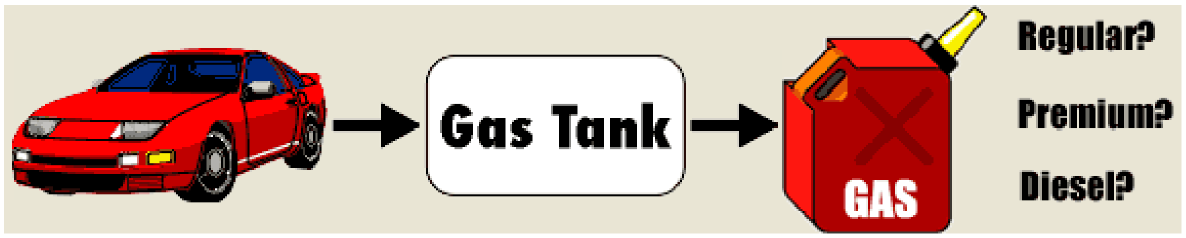
So, what kind of gas are we going to put into the **gastank**? **Regular**, **premium**, **diesel**? Respectively, for each of these options, we would write:

```
car.gastank.fill(regular)
car.gastank.fill(premiun)
car.gastank.fill(diesel)
```

Or, suppose for some reason we wanted to fill our gast tank with a mixture of **all three** types of gas; how would we do that? If we had a **gascan**, we could put some of each of the different grades of gas into that first.

Let's incorporate the concept of **variables** into our program. In this context, a **variable** is like the **gascan**. It's called a variable because its contents can *vary*. That is, we can fill the gascan with **regular**, **premium**, **diesel** or a **mixture of all three**.

Let's see what the JavaScript equivalent of using the variable **gascan** would be:



To fill the car with a single type of gas, such as regular, you would use this syntax:

```
gascan=regular
car.gastank.fill(gascan)
```

To fill the car with three types of gas, you would use this syntax:

```
gascan=regular + premium + diesel
car.gastank.fill(gascan)
```

The **gascan** variable is set equal to **regular** **+** **premium** **+** **diesel**. The variable now **contains** these **values**. Values are any constant entity. The number **1** is a value, and the string **My House** is a value. A variable can be set equal to these values. Say it aloud a couple of times, "variables are containers for values."

Let's apply these concepts to our previous example. In HTML, type this code into the Editor:

CODE TO TYPE:

```
<html>

<body>
<script type="text/javascript">
<!--

beginPhrase = "Your name is "
userName = window.prompt("Enter your name:","")
endPhrase = " Welcome to my page."
finalString = beginPhrase + userName + endPhrase
window.document.write(finalString)

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

</html>
```

Click Preview. The results are the same, but you'll find the code more readable now that it is split into several statements.

All of the words in blue are **variable names**. Each of the first three variables holds the value of the **red text**. In the case of the **userName** variable, the **window.prompt()** method is called first, and the value that is entered by the user is stored in the **userName** variable.

The fourth variable, **finalString**, holds the combined value of each of the previous variables. Then the **write()** method is used to write the value of the **finalString** variable to the document.

When the **finalString** variable was used by the **write()** method, it was not enclosed in quotation marks. That's because we didn't want finalString to be treated as a constant value. Try enclosing **finalString** within quotation marks and see what happens. The text **finalString** will be written to the document. The use of quotation marks within the write() method is limited to text you want to print as a literal string.

One last note about variables before we move on to the next lesson: Variable names must begin with either a letter or the underscore character. You can use any word for a variable name as long as it's not a word reserved by JavaScript, or the name of a JavaScript object, method, or property. If you ever get an error regarding a variable that you do not think should be causing a problem, try changing to name of the variable to something that is **not** one of the reserved words. This rule is also true for objects, which you will learn more about later in the course.

Methods

Welcome back! In this lesson, we'll work toward a deeper understanding of **methods**. **Methods** are functions that belong to or act upon objects. We won't talk about every single method in this course--there are just too many! Instead, we'll make sure you're familiar with methods and the way they work; we want you to have the ability to understand each method as you encounter it, and be able to put it to use for yourself. This particular lesson will cover some of the methods that can be used with the **window** object and the **document** object.

The window Object

window.alert()

You're already familiar with the **alert()** method from earlier lessons. The **alert()** method displays an alert box with a message and an OK button. The syntax looks like this:

OBSERVE:

```
window.alert(message)
```

alert() is a method of the **window** object. It requires one parameter which is the **message** to be displayed in the box. The value of the message can be entered one of two ways:

- Within quotation marks: For example, you could use **"something you want to say."** The message is displayed in the alert box as-is. That is, **something you want to say** is displayed in the box.
- Without quotation marks: In this case, the value is treated as a variable name. You must assign the value equal to a string elsewhere in your code. For example, you can use the statement:

  message = "something you want to say"
  window.alert(message)

  to display **something you want to say** in the box.

In HTML mode, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>
<head>
<script type="text/javascript">
<!--

value = "This O'Reilly thing is GREAT!"
window.alert(value)

//-->
</script>
</head>
</html>
```

Click Preview.

At this point, you're familiar with the alert() method. Let's take a look at some other methods of the **window** object, starting with **open()**:

The open Method

This method is used to **open()** a new window where you can display additional information to your visitors. The syntax looks like this:

OBSERVE:

```
localname = window.open(URL,name,options)
```

The **open()** method requires three parameters. The first parameter gives the **URL** to be displayed in the new window. The second parameter gives the new window a **name** (This name is important if you need to refer to an object within this window later in your code). The third parameter allows you to control the appearance of the new window. There are many **options** you can choose within the third parameter, like the height and width of the new window, whether it's resizable, has scrollbars, a toolbar, or a menubar. The next example will give you a clearer picture of how these parameters work.

The result of the **open()** method will be stored in the **localname** variable. This defines a local name for the new window. In other words, it stores the window into a variable. This part isn't absolutely necessary, but, as you'll see, it can make for more efficient scripting later.

Our example will use an event handler to call a function. Functions will be discussed in greater detail later. For now, just focus on the statements that are being executed within the function. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

function openit(){

   winda = window.open("http://www.oreillyschool.com/cert/javascript1/udidit.htm
l","newone","width=620,height=400")

}

//-->
</script>

</head>

<body>

<form>
   <input type="button" onClick="openit()" value="open a window" />
</form>

</body>
</html>
```

Click Preview.

WARNING Do not put return characters in the statement with the open() method! JavaScript considers each line to be a complete command. Try putting a carriage return in the code (after **width=620** for instance) and see for yourself.

The focus Method

Did the new window that you opened in the last example appear in front of all the other windows that are open on your computer? The **focus()** method will ensure that this is always the case! In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

function openit(){

winda = window.open("http://www.oreillyschool.com/cert/javascript1/udidit.html",
"newone","width=620,height=400")
winda.focus()

}

//-->
</script>

</head>

<body>

<form>
   <input type="button" onClick="openit()" value="open a window" />
</form>
</body>
</html>
```

Click Preview.

Notice that you used the localname **winda** to specify which window would be in **focus**.

The document Object

Theses methods are all methods of the **document** object:

document.open()
document.write()
document.close()

These three methods are ofen used together, so we may as well learn about them all at once!

Suppose you want to open a new window, but don't want to place a pre-existing web page in the window. Instead, you just want to display a quick message. To do that, you will have to write to the new document from the original page. In HTML mode, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

function openit(){

   winda = window.open("","newone","toolbar=no,width=200,height=200")
   winda.focus()
   winda.document.open()
   winda.document.write("IT WORKS DUDE!")
   winda.document.close()

}

//-->
</script>

</head>

<body>

<form>
   <input type="button" onClick="openit()" value="open a window" />
</form>

</body>
</html>
```

Click Preview.

Notice that you did not supply a **URL** when you opened the new window this time. You left this parameter empty, and used **""** instead.

In this example, the words **IT WORKS DUDE!** were written to the new window. **You can write any HTML tags to the new window as well.** Be careful with return characters! Try adding a carriage return after one of the words in **IT WORKS DUDE!** and see happens.

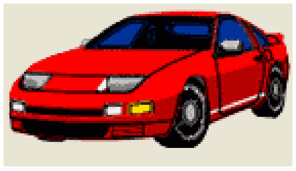
You *must* **open()** the document before you can **write()** to the document that will appear in the new window. When you're finished, you *must* **close()** it. If you don't, your code will produce errors!

Try to write() some HTML tags to the new window to make it more beautiful!

Properties and the Document Object Model

The Document Object and Its Properties

So far in the course, we've introduced the document object and its methods. In this lesson we'll learn about **properties**. Properties are the characteristics of an object. Let's reconsider the car we used in an example earlier:



One distinguishing characteristic of a car is its color. If we wanted to access the color property of a car, we would use this syntax:

car.color

If we wanted to set the color of the car equal to an actual value, we would use this syntax:

car.color="red"

We can also access and set properties of HTML objects. Let's do that now. In HTML mode, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<head>
<title>
Your title
</title>
</head>
<body bgcolor="white">
<script type="text/javascript">
<!--

alert(window.document.title)

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

Click Preview. You'll see an alert in which the **title** of your document is displayed.

The **window** object has no properties; it only has sub-objects. **document** is a sub-object of window. It is used to retrieve properties of the document, and to examine and modify the HTML elements and text within the document. As its name implies, the **title** property is used to access the title of the document object.

Try altering your script to access these properties of the document object:

- bgColor
- linkColor
- vlinkColor

Cool, huh? We can also use JavaScript to set these values. In HTML mode, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<body bgColor="white">
<script type="text/javascript">
<!--
window.document.bgColor = "blue"
//-->
</script>
</body>
</html>
```

Click Preview. Even though you set the **bgColor** attribute of the body tag to **white**, the background color of the page is actually **blue**. That's because you reset the **bgColor** property of the **document** to **"blue"** using JavaScript.

Okay, now try doing the same thing using the **title** property; change the text displayed in the title bar to something else.

Did it work? Well, that all depends on which browser you're using. Not all of the properties that can be accessed with JavaScript can be set with JavaScript. In addition, Internet Explorer's JScript doesn't always produce the same results as other versions of JavaScript. The moral of the story is to test your scripts on *both browsers* whenever you can!

The Document Object Model

The HTML **Document Object Model** (**DOM**) was born out of necessity. As client-side web programming languages evolved, so did the way browsers implemented them. This often produced pages that looked radically different depending on which browser was being used to view the page.

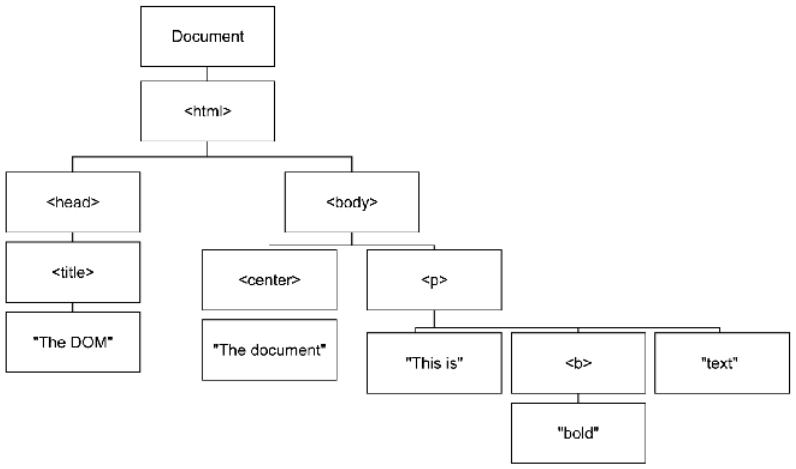
An organization called the World Wide Web Consortium (W3C) was created to help develop common protocols and ensure interoperability on the internet. Part of their work has been to define the HTML DOM.

A DOM is an application programming interface (API) for representing documents (such as HTML documents); it allows access and manipulation of the various elements (such as HTML tags and strings of text) that make up that document. The HTML DOM is rooted in the hierarchical structure of HTML documents. It is represented in the DOM as a tree structure.

OBSERVE:

```
<html>

<head>
   <title>The DOM</title>
</head>

<body>
   <center>The document</center>
   <p>
      This is <b>bold</b> text
   </p>
</body>
</html>
```

The DOM representation of this document in the tree structure:



The node directly above a node is the **parent** of that node. The node directly below a node is the **child** of that node. You could also visualize this hierarchy like this.

In the next lesson, we'll continue our discussion of the DOM and how to access and manipulate HTML objects. For now, just center yourself and visualize HTML documents as hierarchical...

Document Object Methods: getElementById

In the last lesson, you learned how to access and set some properties of the document object. In this lesson, you'll learn how to access and set properties of other objects within your document, using the **getElementById()** method. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<body>

<span id="someText" style="position:absolute; top:100; left:120; background-color:red;
color:white;">
   Here is a some text
</span>

<script type="text/javascript">
<!--

var someText = document.getElementById("someText")
alert(someText)

//-->
</script>

</body>
</html>
```

Click Preview. You'll see an alert displaying the text **[object]**. While this information might seem kind of vague, it can actually be used to gather much more specific information about the element.

The **getElementById()** method returns a reference to the first object with the specified ID. Once you have this reference, you can get more information about the object. For example, you could retrieve the **tagName** of the element. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<body>

<span  id="someText" style="position:absolute; top:100; left:120; background-color:red;
 color:white;">
   Here is some text
</span>

<script type="text/javascript">
<!--

var someText = document.getElementById("someText").tagName
alert(someText)

//-->
</script>

</body>
</html>
```

Click Preview.

Using the getElementById() method, you can access and set the style object properties.

The Style Object Properties

As the web pages you build become more complex, you'll need to access and change the style properties of elements more often. You'll do that using this syntax:

document.getElementById("id").style.styleProperty

The getElementById() method returns a reference to the first object with the specified ID. Then the **style** object is used to access the properties of this object.

Let's try an example to see how this works. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>
<body>

<span id="someText" style="position:absolute; top:100; left:120; background-colo
r:red; color:white;">
   Here is some text
</span>

   <script type="text/javascript">
   <!--

   var someText = document.getElementById("someText").style.top
   alert(someText)

   //-->
   </script>

</body>
</html>
```

Click Preview. You'll see an alert displaying the top position of the element.

Now change the code to obtain some other property, like the left position or background color of the object. Check out this reference to see a list of available properties.

Changing Property Values

Now that you know how to access properties of the style object, let's use JavaScript to change some of those properties. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

function changeBackground(){

   document.getElementById("someText").style.backgroundColor = "blue"
}

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

<body>

<span id="someText" style="position:absolute; top:100; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground()">
      Here is some text
</span>

</body>
</html>
```

Click Preview. The color of the background should change from red to blue when you roll your cursor inside of the box. Go ahead and change some of the other properties, such as the position properties or the color of the text.

Later in the course we'll learn how to change properties of an object to create more complex projects, like dynamic pull-down menus. To do that, you'll need to become familiar with writing your own functions. Fortunately, we're covering functions in the next lesson! See you in the there!

Functions

So, we've created scripts that will be executed when a webpage is loaded into a browser, explored about event handlers, and learned how to use events to trigger your script. In this lesson you'll learn how to create and use **functions** so that your script can be executed at specific times.

A **function** is a self-contained section of code that performs operations on some input and returns a result. Once you define a function, it can be called in much the same way as you'd call a method and it can be called as many times as you'd like.

Let's get to work on an example. In HTML, type this into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function changeBackground(){

   document.getElementById("someText").style.backgroundColor = "blue"
}


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

<body>

<span id="someText" style="position:absolute; top:100; left:120; background-color:red;
color:white"
   onmouseover="changeBackground()">
      Here is a some text
</span>

</body>
</html>
```

Click Preview. I hope thiis looks familiar to you--it's the exact code from an example in the previous lesson. When you move your **mouse over** the element, the **changeBackground** function is called and the background color of the element changes from red to blue.

Note The placement of the function in the code. Functions, like all JavaScript, must be located inside script tags, usually in the head of the document. I encourage you to follow this practice, even though you can place functions just about anywhere.

Functions must be defined using **function functionName()**. You can name your functions anything you like, but it's a good idea to choose a name that describes the script's action. In this case, we named the function **changeBackground** because the script changes the background color of an element. Remember, function names are case-sensitive, so make sure the name you use to call the function is exactly the same as the one you use to define it.

The name of the function will always be followed by a set of **parentheses**. Items placed within these parentheses are called **parameters**. Parameters are passed to the function when it's called. They are values that are used by variables within the function. In our current example, we are not passing any parameters, so the parentheses are empty.

In our example, the contents of the function are placed within a set of curly brackets. An open bracket **{** tells the browser that we are beginning to define the contents of our function. A closing bracket **}** tells the browser that we've completed the function definition.

Using Multiple Functions

As your code becomes more complex, you'll need to create and use more than one function. Let's add a function to our code so that the background color of the element will change back to its original color. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function changeBackground(){

   document.getElementById("someText").style.backgroundColor = "blue"
}

function changeBackgroundBack(){

   document.getElementById("someText").style.backgroundColor = "red"
}


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

<body>

<span id="someText" style="position:absolute; top:100; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground()" onmouseout="changeBackgroundBack()">
      Here is a some text
</span>

</body>
</html>
```

Click Preview. Move your mouse in and out of the element and see the colors change!

This time we're calling a new function called **changeBackgroundBack** when the **mouse is moved out** of the element.

Function Parameters

Suppose you want to create a web page with several boxes that function just like the one in the previous example. One way to do it would be to create two new functions for this box. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function changeBackground1(){

   document.getElementById("someText").style.backgroundColor = "blue"
}

function changeBackgroundBack1(){

   document.getElementById("someText").style.backgroundColor = "red"
}


function changeBackground2(){

   document.getElementById("moreText").style.backgroundColor = "blue"
}

function changeBackgroundBack2(){

   document.getElementById("moreText").style.backgroundColor = "red"
}

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

<body>

<span id="someText" style="position:absolute; top:100; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground1()" onmouseout="changeBackgroundBack1()">
      Here is a some text
</span>

<span id="moreText" style="position:absolute; top:130; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground2()" onmouseout="changeBackgroundBack2()">
      Here is more text
</span>

</body>
</html>
```

Click Preview, then move your mouse in and out of the elements to make sure the colors change.

Works great! But our code will be long and inefficient if we keep building it like this, especially if we want to add more elements that will function in the same way. Instead, we can use parameters to shorten the code. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function changeBackground(id){

   document.getElementById(id).style.backgroundColor = "blue"
}

function changeBackgroundBack(id){

   document.getElementById(id).style.backgroundColor = "red"
}

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

<body>

<span id="someText" style="position:absolute; top:100; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground('someText')" onmouseout="changeBackgroundBack('
someText')">
      Here is a some text
</span>

<span id="moreText" style="position:absolute; top:130; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground('moreText')" onmouseout="changeBackgroundBack('
moreText')">
      Here is more text
</span>

</body>
</html>
```

Click Preview. You'll see the same results as before.

The code in this example is much shorter than the code used in the previous one. To shorten it, we wrote it so the function requires a parameter. When a parameter is passed to our function, the value will be stored in the parameter, much like a variable. This value will be used anywhere you see the parameter name inside the function. So, when we execute the function call:

onmouseover="changeBackground('someText')"

'someText' is passed to the function and is stored in **id** where it will be used in this statement:

document.getElementById(id).style.backgroundColor = "blue"

Multiple Parameters

It is also possible to pass multiple parameters to your functions. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function changeBackground(id,theColor){

   document.getElementById(id).style.backgroundColor = theColor
}

function changeBackgroundBack(id,theColor){

   document.getElementById(id).style.backgroundColor = theColor
}

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

<body>

<span id="someText" style="position:absolute; top:100; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground('someText', 'blue')" onmouseout="changeBackgrou
ndBack('someText', 'red')">
      Here is a some text
</span>

<span id="moreText" style="position:absolute; top:130; left:120; background-colo
r:red; color:white"
   onmouseover="changeBackground('moreText', 'yellow')" onmouseout="changeBackgr
oundBack('moreText', 'red')">
      Here is more text
</span>

</body>
</html>
```

Click Preview. This time when the mouse is moved over the second element, the background should change to **'yellow'** instead of **'blue'**.

Being able to create and use functions is important for JavaScript programmers, so you'll want to be sure you understand the above examples. Experiment! What other functions can you create?

Operators

Operators in Javascript

An **operator** is a symbol used to perform an operation on some value. In JavaScript, operators can act on either numbers or strings. The operators fall into the following categories: arithmetic, assignment, and comparison

The Arithmetic Operators

Arithmetic operators are probably most familiar to you. Let's take a look at the addition (**+**), subtraction (**-**), multiplication (*****), and division (**/**) operators in action. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

var topPosition = 200

function changeIt(){
   topPosition = topPosition + 100
   document.getElementById("someText").style.top = topPosition
}

//-->
</script>

</head>

<body>

<span id="someText" style="position:absolute; top:200;">
   Here is some text
</span>
<br/><br/>
<form>
   <input type="button" onClick="changeIt()" Value="Click Me!" />
</form>

</body>
</html>
```

Click Preview. When you click the button, 100 is **added** to the **topPosition** variable. The top position of the element is set to this value, so the text moves down 100 pixels.

Note

If you are viewing this example using Firefox you must use:

document.getElementById("someText").style.top = topPosition + 100 + "px"

Firefox requires that all position properties be specified as pixel (px) or percentage (%) values.

So, what do you think would happen if you **subtracted** 100 from the top position? If you **multiplied** the top position by 5? Or if you **divided** the top position by 10? Try some experimenting and find out!

Another arithmetic operator, the **modulus** operator (**%**), is used to find the remainder of a division. Since any even number divided by 2 gives no remainder, you can use this operator to determine whether or not a number is odd or even. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

function findIt(){
   var oddOrEven = document.getElementById("myText").value % 2
   alert(oddOrEven)
}

//-->
</script>

</head>

<body>
Enter a whole number in text field below.
<br/>
<form>
   <input type="text" id="myText" />
   <input type="button" onClick="findIt()" Value="Click Me!" />
</form>

</body>
</html>
```

Click Preview. The alert box should display **0** if the number entered is even, and **1** if the number entered is odd.

Finally, we have the **increment** (**++**) and **decrement** (**--**) operators. These operators are used to increase or decrease a value by 1. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

var topPosition = 200

function changeIt(){
   topPosition++
   document.getElementById("someText").style.top = topPosition
}

//-->
</script>

</head>

<body>

<span id="someText" style="position:absolute; top:200;">
   Here is some text
</span>
<br/><br/>
<form>
   <input type="button" onClick="changeIt()" Value="Click Me!" />
</form>

</body>
</html>
```

Click Preview. Be sure to click the button repeatedly--the element should move down by one pixel each time. That's because the increment operator **increases** the top position of the element by 1 each time the function is executed.

What do you think would happen if you used the decrement operator to **decrease** the top position of the element by 1? Give that a try!

The Assignment Operators

The assignment operators are used to set values. For that reason, all of the assignment operators use an **equals** (**=**) sign. We have used this operator before. Remember this function from the first example in this lesson?:

OBSERVE:

```
function changeIt(){
   topPosition = topPosition + 100
   document.getElementById("someText").style.top = topPosition
}
```

Here you used the **=** to assign the position of the element **someText** to a new value. You also could have done this using the **+=** operator. Modify your code as shown:

CODE TO TYPE:

```
<html>
<head>

<script type="text/javascript">
<!--

var topPosition = 200

function changeIt(){
   topPosition += 100
   document.getElementById("someText").style.top = topPosition
}

//-->
</script>

</head>

<body>

<span id="someText" style="position:absolute; top:200;">
   Here is some text
</span>
<br/><br/>
<form>
   <input type="button" onClick="changeIt()" Value="Click Me!" />
</form>

</body>
</html>
```

Click Preview. You'll get the exact result that you did when you viewed the first example from the lesson.

The expression **x += y** is equivalent to the expression **x = x + y**. Both expressions yield the same result. Here's a table illustrating the other assignment operators:

Assignment Operator	Equivalent Expression
x += y	x = x + y
x -= y	x = x - y

x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

Experiment freely with these operators!

Comparison Operators

Comparison operators are used to compare two values. You won't use these operators until the next lesson, but here's a table you can use as a reference later in the course:

Comparison	Operator	Example
equal	==	var x = 3 var y = 4 x == y returns false
not equal	!=	var x = 3 var y = 4 x != y returns true
greater than	>	var x = 3 var y = 4 x > y returns false
less than	<	var x = 3 var y = 4 x < y returns true
greater than or equal to	>=	var x = 3 var y = 4 x >= y returns false
less than or equal to	<=	var x = 3 var y = 4 x <= y returns true

Operators are an important part of JavaScript; you'll use them regularly as you continue through the course and whenever you program in Javascript. Make sure you understand the concepts discussed here thoroughly before moving on to the next lesson.

The if Statement

The if Statement

The **if** statement is one of the most powerful tools at your disposal. It allows you to make comparisons and react to the result of those comparisons. The **if** statement is known as a conditional statement because it's used to perform different actions based on different conditions.

Take a closer look. In HTML, type the following into the Editor so it looks like this:

CODE TO TYPE:

```
<html>

<head>

<script type="text/javascript">
<!--

function doIt(){

   var i = Math.ceil(Math.random()*10)
   var oddOrEven = i%2

      if (oddOrEven == 0){
      alert(i + " is an even number.")
   }
}

//-->
</script>

</head>
<body>

<form>
   <input type="button" onclick="doIt()" value="Do It" />
</form>

</body>

</html>
```

Click Preview. Be sure to click the button several times until an alert box is displayed.

When the function is called, two variables are set. First, the **i** variable is set equal to a random integer. The **modulus operator** is then used to find the remainder of this integer when it is divided by 2. The remainder is stored in a variable called **oddOrEven**.

Keep in mind that when an even integer is divided by 2, there will be no remainder. When an odd integer is divided by 2, the remainder is 1. And our **if** statement can test for just that!

An **if** statement must contain these three elements: **if**, **parentheses**, and **a set of curly brackets**. The word **if** is used to designate the beginning of the **if** statement. The **parentheses** are used to contain the condition to be evaluated. The **curly brackets** contain the code that should be executed **if** the condition evaluates to true.

In this case, the condition uses the **equality** operator to determine whether the **oddOrEven** variable is equal to **0**. If it is equal to 0, then there was no remainder. We know that **i** is an even number; the alert will tells us.

What will happen if the variable is odd? For now, nothing! Let's add code to fix that. In HTML, modify your code as shown:

CODE TO TYPE:

```
<html>

<head>

<script type="text/javascript">
<!--

function doIt(){

   var i = Math.ceil(Math.random()*10)
   var oddOrEven = i%2

   if (oddOrEven == 0){
      alert(i + " is an even number.")
   }

   else{
      alert(i + " is an odd number.")
   }
}

//-->
</script>

</head>
<body>

<form>
   <input type="button" onclick="doIt()" value="Do It" />
</form>

</body>

</html>
```

Click Preview. Again, be sure to click the button several times so that you get a look at the two different alert boxes.

This time, we added an **else** statement to our function. The else statement will be executed whenever the condition in the **if** statement does not evaluate to true. In this case, that means anytime the **oddOrEven** variable contains a value other than 0.

Now suppose you want something special to happen if the variable **i** is equal to 2. To do that, you'll include an **else if** statement. In HTML, modify your code as shown:

CODE TO TYPE:

```
<html>

<head>

<script type="text/javascript">
<!--

function doIt(){

   var i = Math.ceil(Math.random()*10)
   var oddOrEven = i%2

   if (i == 2){
      alert("The number is 2!")
   }

   else if (oddOrEven == 0){
      alert(i + " is an even number.")
   }

   else{
      alert(i + " is an odd number.")
   }
}

//-->
</script>

</head>
<body>

<form>
   <input type="button" onclick="doIt()" value="Do It" />
</form>

</body>

</html>
```

Click Preview. Again, click the button several times so that you can observe all three alert boxes.

Note that in this case, the initial **if** statement checks to see if the **i** variable is equal to **2**. The function **must** be written in this way. What do you think would happen if **i** was equal to 2 and **oddOrEven** variable was checked first? Ponder this for a moment, and experiment at will!

Logical Operators

Logical operators are used to link sequences of comparisons. You can use them to check two conditions in a single **if** statement. Two of the more common logical operators are the **AND** operator and the **OR** operator.

We'll try an example. In HTML, type the code as shown:

CODE TO TYPE:

```
<html>

<head>

<script type="text/javascript">
<!--

function doIt(){

   var i = Math.ceil(Math.random()*10)
   var oddOrEven = i%2

   if (oddOrEven == 0 && i > 6){
      alert(i + " is an even number.  It is greater than 6.")
   }

   else if (oddOrEven == 0 && i <= 4){
      alert(i + " is an even number.  It is less than or equal to 4.")
   }

   else if (oddOrEven == 0 && i == 6){
      alert(i + " is an even number.  It is equal to 6.")
   }

   else{
      alert(i + " is an odd number.")
   }
}

//-->
</script>

</head>
<body>

<form>
   <input type="button" onclick="doIt()" value="Do It" />
</form>

</body>

</html>
```

Click Preview. As before, be sure to click the button several times.

Here, we used the **AND** operator (**&&**) to check two conditions simultaneously. That is, we checked to see if the variable was even **AND** whether it was **greater than 6**, **less than or equal to 4**, or exactly equal to **6**.

As your code becomes more complex, you'll use **if** statements more often. Experiment with them a for a while before moving on to the next lesson. See you there!

For and While Loops

for and while Loops

A **loop** is a section of code you want to execute a specific number of times or while a specified condition is true. Here's a real life example:

Suppose you're really hungry, so you sit down to eat some spaghetti. To eat, you perform the same set of actions repeatedly. That is, you use your fork to deliver a small amount of food to your mouth, then you chew the food, and finally you swallow. You continue this process for as long as you're still hungry. That is, you **loop** through this process until you are satisfied.

The two main types of loops used in JavaScript are **while** loops and **for** loops. Let's take a closer look.

The while Loop

A **while** loop is used to execute the same section of code **while** a specified condition is true. In HTML, type this code into the Editor below as shown:

CODE TO TYPE:

```
<html>

<head>

<script type="text/javascript">
<!--

function doIt(){

   var i=0

   while (i<5 ){

      alert("The number is" + i)
      i++

   }
}

//-->
</script>

</head>
<body>

<form>
   <input type="button" onclick="doIt()" value="Do It" />
</form>

</body>

</html>
```

Click Preview. Note that an alert is displayed each time the loop is repeated, and that a total of five alert messages are displayed.

When the button is clicked, the **doIt** function is called, and a variable **i** is **initialized** (or set equal to) **0**. This is an important step because **i** is also used in the **while** loop to **test** and see if a condition is true. As long as value of **i** is less than **5**, the code written within the **curly brackets** will be executed.

Each time this section of code is executed, the value of **i** is **incremented** by **1**. So, after the fifth time the piece of code is executed, the value of **i** will be equal to 5. After that, since 5 is exactly equal to 5, the condition is no longer true and the loop terminates.

The for Loop

The **for** loop is similar to a **while** loop, but a **for** loop will execute a section of code a specified number of times. Take a look. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>

<script type="text/javascript">
<!--

function doIt(){

   for (i=0;i<5;i++){

      alert("The number is" + i)

   }
}

//-->
</script>

</head>
<body>

<form>
   <input type="button" onclick="doIt()" value="Do It" />
</form>

</body>

</html>
```

Click Preview. The result of this code is the same as the result of the previous code.

In the case of a **for** loop, a variable is **initialized**, **tested**, and **incremented** in the same location.

Initially, **i** is equal to **0**. Since **i** is **less than 5** the code written within the **curly brackets** will be executed.

After this code is executed, **i** is **incremented** by **1** and so it is now equal to 1. Since 1 is still **less than 5**, the alert is displayed again, and the value of **i** is **incremented**. This process continues until the value of **i** is no longer less than 5.

Nesting Statements

Now that you're familiar with **while** loops, **for** loops, and **if** statements, let's try using them together. In this example, we'll **nest** an **if** statement within a **for** loop. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>

<script type="text/javascript">
<!--

function doIt(){

   for (i=0;i<5;i++){

      if(i==4){

         alert("The number is 4.  This is the last time through the loop.")

      }

      else{

         alert("The number is" + i)

      }
   }
}

//-->
</script>

</head>
<body>

<form>
   <input type="button" onclick="doIt()" value="Do It" />
</form>

</body>

</html>
```

Click Preview. The result will be similar to the last example, except this time a special message will be displayed the last time the loop is executed.

When **i** is exactly **equal to 4**, an alert box will be displayed to let you know that it's the final time the loop will be executed. If **i** is not equal to 4, a "normal" alert box is displayed.

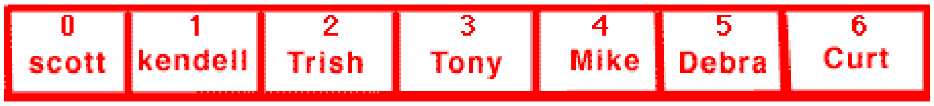
Loops are a powerful tool, one you'll use often in your scripts. Experiment with them for a bit before moving on to the next lesson. See you there!

Object Arrays

Arrays

An **array** is a single variable name that can be used to store a set of values. It can hold a series of values, and the individual values are accessed by their position in the **array**. You can think of an array like a list or a filing cabinet.

An array called **names** might look something like this:



This is the syntax used to reference an element stored in an array:

arrayName[index]

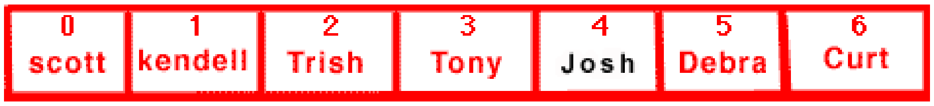
The first name stored in the array is **scott**. If you wanted to access this name, you would do so using:

names[0]

Be aware that the first index of an array is always **0**. So, if you wanted to change the name stored in the fifth position from **Mike** to **Josh**, you would use:

names[4] = "Josh"

The array then becomes:



Arrays will be the focus of the next two lessons. You'll create your own arrays, like the **names** array above, in the next lesson. For now, we're going to learn about the arrays that already exist when you create an HTML document.

document Object Collections

Suppose you had an HTML document that looked like this. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function displayText(){

   document.getElementById("myText").value = document.getElementById("myForm").n
ame

}

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

<body>

<form id="myForm" name="myFormName">
   <input type="text" id="myText" />
   <input type="button" value="Do It" onClick="displayText()" />
</form>

</body>
</html>
```

Click Preview. When you click the button, the name of the form will be displayed in the text field.

At this point, you're familiar with the **getElementById** syntax. You can accomplish the same task that syntax does, using the **forms** collection. Let's give thatt a try. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function displayText(){

   document.getElementById("myText").value = document.forms[0].name

}

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

<body>

<form id="myForm" name="myFormName">
   <input type="text" id="myText" />
   <input type="button" value="Do It" onClick="displayText()" />
</form>

</body>
</html>
```

Click Preview. You'll see the same results that you did earlier using syntax.

The forms collection is just one of the **document object collections** available in Javascripts. To learn about the others, click **here**. For our purposes in this lesson, we'll focus on the forms collection.

The forms Collection

All inputs, textareas, and select boxes are considered elements that are contained by a form. As such, they can be referenced using the **elements** array. In HTML, type the code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function displayText(){

   document.getElementById("myText").value = document.forms[0].elements[0].id

}

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

<body>

<form name="myForm">
   <input type="text" id="myText" />
   <input type="button" value="Do It" onClick="displayText()" />
</form>

</body>
</html>
```

Click Preview. The first element in the form is the **text input**. Because it's the first element, it can be referenced using the **0** index.

Let's add another element within the form. In HTML, modify your code in the Editor below as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function displayText(){

   document.getElementById("myText").value = document.forms[0].elements[1].name

}

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

<body>

<form id="myForm" name="myFormName">
   <input type="text" id="myText" />
   <select name="mySelect">
      <option>Option 1</option>
      <option>Option 2</option>
      <option>Option 3</option>
   </select>
   <input type="button" value="Do It" onClick="displayText()" />
</form>

</body>
</html>
```

Click Preview. You'll see the **name** of the **select** box displayed. Since it's the second element in the form, it's referenced using the **1** index.

The select box is a special element because it has **options** associated with it. These options are considered elements contained by the select box, so they can be referenced using the **options** array. In HTML, modify your code as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function displayText(){

   document.getElementById("myText").value = document.forms[0].elements[1].optio
ns[0].text

}

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

<body>

<form id="myForm" name="myFormName">
   <input type="text" id="myText" />
   <select name="mySelect">
      <option>Option 1</option>
      <option>Option 2</option>
      <option>Option 3</option>
   </select>
   <input type="button" value="Do It" onClick="displayText()" />
</form>

</body>
</html>
```

Click Preview. You'll see the text **Option 1** displayed in the text field. This text is contained by the first **option**, so it's referenced using the **0** index.

table Collections and innerHTML

Tables have two collections that can be used to change the content of a table dynamically. The **rows** collection can be used to return an array containing each row from a table. The **cells** collection can be used to return an array containing each cell from a table.

To get a better idea of how these collections work, we'll use the **innerHTML** property. The **innerHTML** property sets or retrieves the HTML between the start and end tags of the object.

In HTML, modify your code as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

function displayRowInfo(){

   alert(document.getElementById("theTable").rows[2].innerHTML)

}

function displayCellInfo(){

   alert(document.getElementById("theTable").rows[2].cells[0].innerHTML)

}

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

<body>

<table id="theTable" border="2">
   <tr>
      <td>Row1, cell1</td>
      <td>Row1, cell2</td>
   </tr>
   <tr>
      <td>Row2, cell1</td>
      <td>Row2, cell2</td>
   </tr>
   <tr>
      <td>Row3, cell1</td>
      <td>Row3, cell2</td>
   </tr>
</table>

<br/>

<form name="myForm">
   <input type="button" value="Row Info" onClick="displayRowInfo()" />
   <input type="button" value="Cell Info" onClick="displayCellInfo()" />
</form>

</body>
</html>
```

Click Preview. Be sure to click both buttons on the webpage.

Clicking the first button will alert the **innerHTML** of the third row. Because it's the third row, you can access it using **rows[2]**.

Clicking the second button will alert the **innerHTML** of first cell in the third row. Because it's the first cell, you can access it using **cells[0]**.

Note

As your scripts become more complex, you might find it helpful to adopt syntax that uses variables within your references. For example, you could also access the **innerHTML** of the first cell in the third row using:

var cell1=document.getElementById("theTable").rows[2].cells[0]
alert(cell1.innerHTML)

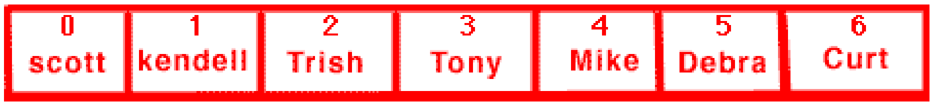
In the next lesson we'll discuss arrays in more detail. Be sure to experiment with what you learned here before moving on!

Arrays

More Arrays

In the last lesson, you learned about arrays and used them along with several object collections available via the document object model. In this lesson, you'll define your own object arrays to use in your own JavaScripts!

Recall that an **array** is a single variable name that can be used to store a set of values. Let's look at the example from the previous lesson:



The first name stored in the array is **scott**. If you wanted to access this name you would do so using:

names[0]

Let's build this array in our script, and access one of the values stored within it. In HTML, type this code into the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

var names = new Array()

names[0] = "scott"
names[1] = "kendell"
names[2] = "Trish"
names[3] = "Tony"
names[4] = "Mike"
names[5] = "Debra"
names[6] = "Curt"

function displayText(){

   alert(names[0])

}

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

<body>

<form id="myForm">
   <input type="button" value="Do It" onClick="displayText()" />
</form>

</body>
</html>
```

Click Preview. Be sure to click on the button, which calls the **displayText** function, which in turn will display **names[0]** in an alert box. **scott** is displayed, because it is the **0** index of the **names** array.

To define your own array, you must first create an **Array** object; use the **new** keyword to do that.

There are two ways to add values to an array. One is to add a single value per line, as we did here:

names[0] = "scott"
names[1] = "kendell"
names[2] = "Trish"
names[3] = "Tony"
names[4] = "Mike"
names[5] = "Debra"
names[6] = "Curt"

Alternatively, you can add values using a single line of code. In HTML, modify your code in the Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

   var names = new Array("scott","kendell","Trish","Tony","Mike","Debra","Curt")

function displayText(){

   alert(names[0])

}

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

<body>

<form id="myForm">
   <input type="button" value="Do It" onClick="displayText()" />
</form>

</body>
</html>
```

Click Preview. You get the same results as you did in the previous example. This time though, we assigned values to the **names** array when we declared it as a **new Array** object.

Let's experiment with the names array some more! In HTML, modify your code as shown:

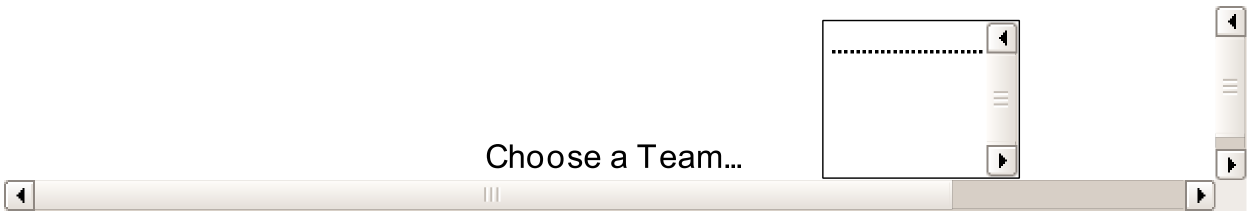
CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

var names = new Array()

names[0] = "scott"
names[1] = "kendell"
names[2] = "Trish"
names[3] = "Tony"
names[4] = "Mike"
names[5] = "Debra"
names[6] = "Curt"

function displayText(){

   alert(names[document.getElementById("myText").value])

}

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

<body>

<form id="myForm">
   <input type="text" id="myText" />
   <input type="button" value="Do It" onClick="displayText()" />
</form>

<br/>
Please enter an integer, 0 through 6.

</body>
</html>
```

Click Preview. Now try entering different integers! Note that the index of the array will be the **value** we enter into the text field.

Building Dynamic Select Boxes

Suppose you want to create a web page that includeds something like this:



Try selecting different teams to see how the player names change. We are **dynamically populating** the second select box based on the choice selected in the first select box. Even if you're not a rabid basketball fan, this is a good example of how a programmer might use arrays in the real world.

Let's start by creating the select boxes. In HTML, type this code into your Editor as shown:

CODE TO TYPE:

```
<html>

<body>

<form>
   <select id="theTeams">
      <option> Choose a Team</option>
      <option>Boston Celtics</option>
      <option>LA Lakers</option>
      <option>Chicago Bulls</option>
    </select>
   <select id="thePlayers" size="3">
      <option> ...............</option>
      <option></option>
      <option></option>
      </select>
</form>

</body>

</html>
```

Click Preview.

This code probably looks familiar to you. You have two select boxes. One contains a list of three teams from which you can choose. The other is empty for now, though it does have one option of **.............** This is a sort of place holder to keep the select box wide enough--it would collapse otherwise. We also have three other empty **<options>** listed. This is important: **You must have as many options as the longest list with which you want to populate the select box**.

Next, you need to make an array to store the names that will be displayed upon selecting a team. This array is called **teams** and contains four arrays -- **teams["none"], teams["celtics"], teams["lakers"]**, and **teams["bulls"]**. In HTML, modify your code as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

var teams = new Array("none","celtics","lakers","bulls")

teams["none"] = new Array(".......................","","")
teams["celtics"]= new Array("Larry Bird","Bill Russell","Kevin McHale")
teams["lakers"] = new Array("Wilt Chamberlain","Jerry West","Shaquille ONeil")
teams["bulls"] = new Array("Micheal Jordan","Scottie Pippen","Dennis Rodman")

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

<body>

<form>
    <select id="theTeams">
        <option> Choose a Team</option>
        <option>Boston Celtics</option>
        <option>LA Lakers</option>
        <option>Chicago Bulls</option>
    </select>
    <select id="thePlayers" size="3">
        <option> ...............</option>
        <option></option>
        <option></option>
    </select>
</form>

</body>

</html>
```

Now you'll write the function that will display the appropriate names based on the selected team. In order to create this function, you need to understand how to reference the teams arrays using numeric indices.

Try to predict which values would be returned if you used these alerts with the above code. (If you're having trouble, you can always insert them one at a time, just before the ending script tag):

```
alert(teams[1])
alert(teams[teams[1]][2])
alert(teams["celtics"])
alert(teams["celtics"][3])
```

You may recall that the **selectedIndex** of a select box returns an integer based upon the highlighted choice. Because you can also access array values using numeric indices, we can use the **selectedIndex** property to display the appropriate values in the names array based upon the highlighted team. In HTML, type the code into your Editor as shown:

CODE TO TYPE:

```
<html>

<head>
<script type="text/javascript">
<!--

var teams = new Array("none","celtics","lakers","bulls")

teams["none"] = new Array(".......................","","")
teams["celtics"]= new Array("Larry Bird","Bill Russell","Kevin McHale")
teams["lakers"] = new Array("Wilt Chamberlain","Jerry West","Shaquille ONeil")
teams["bulls"] = new Array("Micheal Jordan","Scottie Pippen","Dennis Rodman")

function fillPlayers(){

   var whichIndex = document.getElementById("theTeams").selectedIndex
   var numberOfPlayers = teams[teams[whichIndex]].length

   for(i=0;i < numberOfPlayers; i++){
      document.getElementById("thePlayers").options[i].text = teams[teams[whichI
ndex]][i]
   }
}

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

<body>

<form>
    <select id="theTeams" onChange="fillPlayers()">
        <option> Choose a Team</option>
        <option>Boston Celtics</option>
        <option>LA Lakers</option>
        <option>Chicago Bulls</option>
    </select>
    <select id="thePlayers" size="3">
        <option> ...............</option>
        <option></option>
        <option></option>
    </select>
</form>

</body>

</html>
```

Click Preview. You'll see that the code works exactly like the example from earlier in the lesson. Let's take a closer look at the code.

The **fillPlayers** function is called **onChange** of the select box. When the function is called, the following values are returned:

var whichIndex = document.getElementById("theTeams").selectedIndex
var numberOfPlayers = teams[teams[whichIndex]].length

The first line retrieves the selectedIndex of theTeams select box, which will be an integer between 0 and 3. This value is stored in the **whichIndex** variable.

The second line retrieves the **length** of the array stored in **teams[teams[whichIndex]]**. Suppose **whichIndex** is equal to **1**. Then **teams[whichIndex]** is actually **teams[1]**, otherwise known as **celtics**. Since the **teams["celtics"]** array has three items, the **numberOfPlayers** would be set equal to **3**.

The **for loop** is used to populate the select box:

```
for(i=0;i < numberOfPlayers; i++){
document.getElementById("thePlayers").options[i].text = teams[teams[whichIndex]][i]
}
```

If **numberOfPlayers** is set equal to 3 (and it will be no matter which team is selected since all of the arrays contain three items), then the code within the for loop will be executed three times. Each time this statement:

```
document.getElementById("thePlayers").options[i].text = teams[teams[whichIndex]][i]
```

is executed, the **text** of one of the **options** in **thePlayers** select box is written. The text will be the value stored in **teams[teams[whichIndex]][i]**. If **whichIndex** is set equal to **1** and **i** is set equal to **0**, then the text written to the first option would be **teams[teams[1]][0]**, otherwise known as **Larry Bird**.

Let's pratice--here is another way you can code the above example.

In HTML, type the following into the Editor below:

```
<html>

<head>
<script type="text/javascript">
<!--

var teams = new Array()

teams[1] = new Array()
teams[1][0] = "Larry Bird"
teams[1][1] = "Bill Russell"
teams[1][2] = "Kevin McHale"

teams[2] = new Array();
teams[2][0] = "Wilt Chamberlain"
teams[2][1] = "Jerry West"
teams[2][2] = "Shaquille Oneil"

teams[3] = new Array()
teams[3][0] = "Micheal Jordan"
teams[3][1] = "Scottie Pippen"
teams[3][2] = "Dennis Rodman"


function fillPlayers(){

   var whichIndex = document.getElementById("theTeams").selectedIndex
   var numberOfPlayers = teams[whichIndex].length

   for(i=0;i < numberOfPlayers; i++){
      document.getElementById("thePlayers").options[i].text = teams[whichIndex][
i]
   }
}

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

<body>

<form>
    <select id="theTeams" onChange="fillPlayers()">
        <option> Choose a Team</option>
        <option>Boston Celtics</option>
        <option>LA Lakers</option>
        <option>Chicago Bulls</option>
    </select>
    <select id="thePlayers" size="3">
        <option> ..............</option>
        <option></option>
        <option></option>
    </select>
</form>

</body>

</html>
```

Click Preview.

Do you recognize and understand the above changes? Take time to practice so you can use what you've learned when you do the project for this lesson.

Operations with Nodes

Introduction to Nodes

Every HTML document has a "tree" structure. A tree structure is an embedded heirarchical list of objects. Consider an HTML page that has the following structure:

OBSERVE:

```
<html>
<head>
   <title>Example page</title>
</head>

<body>

   <table name="toptable">
      <tr id="firstrow">
         <td>first cell, first row</td>
         <td>second cell, first row</td>
      </tr>
      <tr id="secondrow">
         <td>first cell, second row</td>
         <td>second cell, second row</td>
      </tr>
      <tr id="thirdrow">
         <td>first cell, third row</td>
         <td>second cell, third row</td>
      </tr>
   </table>

</body>
</html>
```

Just like the above table, HTML forms a hierarchy. The table tag above has **three children** (the three tr tags) and they in turn have two children each (they each have two td tags in them). Click on the button below to see a **Tree View** of the above table in a web page (in fact you'll see a full tree view of the entire page).

Click here

Each of the folders above is a **node** of the HTML DOM tree for this web page. A node is simply where the tree branches. In fact, every HTML element on a webpage is a node. There are two kinds of nodes: **elements** and **text nodes**. Both kinds can be created and manipulated using JavaScript.

Creating and Manipulating Elements

Here is an example using `createTextNode` and `createElement` to create some text and an element. Then using `appendChild` and `insertBefore`, we place the elements and text into the document where we want it:

Type the following into CodeRunner:

```
<html>
<head>
<script type="text/javascript">
<!--
function insertStuff(){

    var theNewElement = document.createElement("b")
    var theNewElement2 = document.createElement("i")
    var theNewText1 = document.createTextNode(" Here is the first text ")
    var theNewText2 = document.createTextNode(" Here is the second text ")
    theNewElement.appendChild(theNewText1)
    theNewElement2.appendChild(theNewText2)

    var here = document.getElementById("scott")
    var parentDiv = here.parentNode

    parentDiv.insertBefore(theNewElement,here)
    parentDiv.appendChild(theNewElement2)

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

<a href="javascript:insertStuff()">Click here to insert the HTML</a>

<p id="scott" style="color:red;">
hello there
</p>

</body>
</html>
```

After you've previewed it, click the link to see **bold** text placed before the red "hello there" and *italic* text placed after it.

Let's break down the code and see what's happening. In the `insertStuff()` function, we declared a new variable called `theNewElement`. This is our container for the new element instance we created using `createElement("b")`, where we created an instance **b** element (the **bold** element) (it's not a *new* element but it is a *new instance* of a familiar element). In the second line we created a new **i** using createElement("i").

After creating a couple of elements, we created a couple of **text nodes** using the `createTextNode("Here is the first text")`.

And since we created a couple of **element nodes** and a couple of **text nodes**, we can append the text nodes as children of the element nodes. We do that by using the **appendChild()** method. The element we are appending to is `theNewElement`, so we write `theNewElement.appendChild(theNewText1)`. The node we are appending goes in the argument of the method (in this case, theNewText1). This operation is the same as forming **<b>Here is the first text</b>** and **<i>Here is the second text</i>**.

Now, the goal is to put the first text in *front* of **hello there** and the second text *after* it. In order to do this, let's create a variable called **here** where we can store **scott** as a **parentNode**. Then we can use the **insertBefore** method to place the element before the p tag, and the **appendChild** method to place the element after the p tag.

Note that the `insertBefore()` method takes one or two parameters. The first parameter is WHAT we are appending, and the second parameter is WHERE we are appending it.

Try playing around with this script and see if you can place the text in different places using the above code. There are several other methods you might want to familiarize yourself with to manipulate nodes: cloneNode, replaceChild, removeChild, createAttribute, and insertData.

Check here for a browser compatiblity chart.

Working with tables

One of the best uses of working with Nodes and children is with tables. You might want to change something in a table or maybe you'd like to add cells or rows dynamically without having to rebuild an entire table. Manipulating Nodes is great when it comes to tables. Try the following code:

Type the following into CodeRunner:

```
<html>
<body>

<table border="1">
<tr id="example">
<td>TD One</td>
<td>TD Two</td>
<td>TD Three</td>
</tr>
</table>

<script type="text/javascript">
<!--
var Counter = 0

function insertTD(){

 var newTD = document.createElement("td")
 var newText = document.createTextNode("A New Cell " + (Counter++))
 newTD.appendChild(newText)

 var tdplace = document.getElementById("example").childNodes[2]
        document.getElementById("example").insertBefore(newTD,tdplace)

}
//-->
</script>

<a href="javascript:insertTD()">Click here</a>

</body>
</html>
```

Okay, now Preview it. When you click on the link, it should insert a cell into the table. In this case we are using the same methods as we did earlier in the lesson. We used `createElement` to create a new td element, then we used `createTextNode` to create a new text node and then `appendChild` to append it to the td element. Finally, we set its location by choosing the `childNodes[2]` of the "example" tr element, which is the third td of the table in its first and only row.

Note

If you are using the Firefox browser you will have to use:

var tdplace = document.getElementById("example").**childNodes[4]**

This is because Firefox will count whitespace as a node. Alternatively you could use:

<tr id="example"><td>TDOne</td><td>TDTwo</td><td>TDThree</td></tr>

Here's another way to do the exact same thing:

Type the following into CodeRunner:

```
<html>
<body>

<table border="1">
<tr id="example">
<td>TD One</td>
<td>TD Two</td>
<td>TD Three</td>
</tr>
</table>

<script type="text/javascript">
<!--
var Counter = 0

function insertTD(){

 var newTD = document.createElement("td")
 var newText = document.createTextNode("A New Cell " + (Counter++))
 newTD.appendChild(newText)

 var trname = document.getElementById("example")
 var tdplace = trname.getElementsByTagName("td").item(2)
 trname.insertBefore(newTD,tdplace)
}
//-->
</script>

<a href="javascript:insertTD()">Click here</a>

</body>
</html>
```

Preview it. You should get the same thing as before. The only difference here is that we are using `getElementsByTagName("td")`. This get's all of the td elements and puts them in a collection. We then can access the third td element in the table by selecting the second item, using `item(2)`.

Here is yet another way to do the same thing.

Change your document to look like this:

```
<html>
<body>

<table border="1">
<tr id="example">
<td>TD One</td>
<td>TD Two</td>
<td>TD Three</td>
</tr>
</table>

<script type="text/javascript">
<!--
var Counter = 0

function insertTD(){

 var newText = document.createTextNode("A New Cell " + (Counter++))

 NewCell = document.getElementById("example").insertCell(2)
        NewCell.appendChild(newText)
}
//-->
</script>

<a href="javascript:insertTD()">Click here</a>

</body>
</html>
```

Now, Preview it. Again, it should do the same thing as before. This time we are using the `insertCell()` method. We don't need to create a td element using createElement since `insertCell()` will do that for us. Once we have the cell created (in the third position, same as before), we append the text node using `appendChild()`.

Phew! I think that's enough for this lesson! Do the projects, pat yourself on the back, and it's on to the next one!

Creating Javascript Menus

Javascript Menus

Welcome to the final lesson of your Javascript course! In this lesson, we'll use what we have learned to create a dropdown menu. These menus will help you create organized and professional paths to navigate through websites. Have fun!

Our first step is to create a simple list of items to navigate. I like dogs, so let's make a list of different types of dogs.

In HTML, type the following into the Editor below:

```
<html>

<body>

<ul id="surf">

  <li>Hounds
    <ul>
      <li><a href="">Bloodhound</a></li>
      <li><a href="">Basset</a></li>
      <li><a href="">Beagle</a></li>
      <li><a href="">Irish Wolfhound</a></li>
    </ul>
  </li>

  <li>Shepherds
    <ul>
      <li><a href="">German Shepherd</a></li>
      <li><a href="">Collie</a></li>
      <li><a href="">Sheepdog</a></li>
    </ul>
  </li>

  <li>Terriers
    <ul>
      <li><a href="">Airedale</a></li>
      <li><a href="">Scottish</a></li>
      <li><a href="">Jack Russell</a></li>
      <li><a href="">Bull Terrier</a></li>
      <li><a href="">Yorkshire</a></li>
    </ul>
  </li>

</ul>


</body>

</html>
```

Preview this code. You should see a list of three types of dogs--Hounds, Shepherds, and Terriers. Listed below each of these are specific breeds of each type. The breed names should be hyperlinks, although clicking on the link will not take you to any specific location.

Because we are creating a dropdown menu, we need to make it so that the three lists of dog breeds are displayed horizontally. While we're at it, let's eliminate the list bullets. We can do all of this using stylesheets.

In HTML, type the following into the Editor below:

```
<html>

<head>
<style type="text/css">

ul {
  padding: 0;
  margin: 0;
  list-style: none;
}

li {
  float: left;
  position: relative;
  width: 10em;
}

</style>
</head>
<body>

<ul id="surf">

  <li>Hounds
    <ul>
      <li><a href="">Bloodhound</a></li>
      <li><a href="">Basset</a></li>
      <li><a href="">Beagle</a></li>
      <li><a href="">Irish Wolfhound</a></li>
    </ul>
  </li>

  <li>Shepherds
    <ul>
      <li><a href="">German Shepherd</a></li>
      <li><a href="">Collie</a></li>
      <li><a href="">Sheepdog</a></li>
    </ul>
  </li>

  <li>Terriers
    <ul>
      <li><a href="">Airedale</a></li>
      <li><a href="">Scottish</a></li>
      <li><a href="">Jack Russell</a></li>
      <li><a href="">Bull Terrier</a></li>
      <li><a href="">Yorkshire</a></li>
    </ul>
  </li>

</ul>


</body>

</html>
```

Preview this code. The position and appearance of the second-level lists must also be considered. That is, these lists should not be displayed until the associated top-level menu item is selected, and they should be displayed in a location so as not to interfere with the main menu.

Let's start by working on the positioning.

In HTML, type the following into the Editor below:

```
<html>

<head>
<style type="text/css">

ul {
  padding: 0;
  margin: 0;
  list-style: none;
}

li {
  float: left;
  position: relative;
  width: 10em;
}

li ul {
  display: block;
  position: absolute;
  top: 1em;
}

</style>
</head>
<body>

<ul id="surf">

  <li>Hounds
    <ul>
      <li><a href="">Bloodhound</a></li>
      <li><a href="">Basset</a></li>
      <li><a href="">Beagle</a></li>
      <li><a href="">Irish Wolfhound</a></li>
    </ul>
  </li>

  <li>Shepherds
    <ul>
      <li><a href="">German Shepherd</a></li>
      <li><a href="">Collie</a></li>
      <li><a href="">Sheepdog</a></li>
    </ul>
  </li>

  <li>Terriers
    <ul>
      <li><a href="">Airedale</a></li>
      <li><a href="">Scottish</a></li>
      <li><a href="">Jack Russell</a></li>
      <li><a href="">Bull Terrier</a></li>
      <li><a href="">Yorkshire</a></li>
    </ul>
  </li>

</ul>


</body>

</html>
```

Click Preview.

A dropdown should only show us the selections when we mouseover or click on a menu item. We need to hide the lists until we mouseover the items. Hiding the lists is easy. Simply change the display of the list items from **block** to **none** as follows.

In HTML, type the following into the Editor below:

```
<html>

<head>
<style type="text/css">

ul {
  padding: 0;
  margin: 0;
  list-style: none;
}

li {
  float: left;
  position: relative;
  width: 10em;
}

li ul {
  display: none;
  position: absolute;
  top: 1em;
}

</style>
</head>
<body>

<ul id="surf">

  <li>Hounds
    <ul>
      <li><a href="">Bloodhound</a></li>
      <li><a href="">Basset</a></li>
      <li><a href="">Beagle</a></li>
      <li><a href="">Irish Wolfhound</a></li>
    </ul>
  </li>

  <li>Shepherds
    <ul>
      <li><a href="">German Shepherd</a></li>
      <li><a href="">Collie</a></li>
      <li><a href="">Sheepdog</a></li>
    </ul>
  </li>

  <li>Terriers
    <ul>
      <li><a href="">Airedale</a></li>
      <li><a href="">Scottish</a></li>
      <li><a href="">Jack Russell</a></li>
      <li><a href="">Bull Terrier</a></li>
      <li><a href="">Yorkshire</a></li>
    </ul>
  </li>

</ul>


</body>

</html>
```

Preview this. The menu listings should now be hidden from sight.

The Script

The next step is to make the listings appear when we move our mouse over the menu items.

In HTML, type the following into the Editor below:

```
<html>

<head>
<style type="text/css">

ul {
  padding: 0;
  margin: 0;
  list-style: none;
}

li {
  float: left;
  position: relative;
  width: 10em;
}

li ul {
  display: none;
  position: absolute;
  top: 1em;
}

li.dropdown ul { display: block; }

</style>

<script type="text/javascript">
<!--

function initializelist(){
           topRoot = document.getElementById("surf");
           for (i=0; i<topRoot.childNodes.length; i++) {
                nodeAt = topRoot.childNodes[i];
                if (nodeAt.nodeName=="LI") {
                    nodeAt.onmouseover=function() {
                         this.className ="dropdown";
                     }
                     nodeAt.onmouseout=function() {
                          this.className="";
                     }
             }
       }
}



window.onload=initializelist;

//-->
</script>

</head>
<body>

<ul id="surf">

  <li>Hounds
    <ul>
      <li><a href="">Bloodhound</a></li>
      <li><a href="">Basset</a></li>
      <li><a href="">Beagle</a></li>
      <li><a href="">Irish Wolfhound</a></li>
    </ul>
  </li>
  <li>Shepherds
    <ul>
      <li><a href="">German Shepherd</a></li>
      <li><a href="">Collie</a></li>
      <li><a href="">Sheepdog</a></li>
    </ul>
  </li>

  <li>Terriers
    <ul>
      <li><a href="">Airedale</a></li>
      <li><a href="">Scottish</a></li>
      <li><a href="">Jack Russell</a></li>
      <li><a href="">Bull Terrier</a></li>
      <li><a href="">Yorkshire</a></li>
    </ul>
  </li>

</ul>


</body>

</html>
```

Preview this. Be sure to move your mouse over each menu item. When you do, it should display the dog breeds belonging to that particular group.

Let's take a closer look at the **initializelist** function. This function will examine all of the childNodes inside of the element called **surf**. If the childNode is an li element, its className will be set equal to **dropdown** when the mouse moves over it. When the mouse moves away, the className will be set equal to "" (a blank name like it had in the beginning).

Observe:

```
function initializelist(){
          topRoot = document.getElementById("surf");
          for (i=0; i<topRoot.childNodes.length; i++) {
               nodeAt = topRoot.childNodes[i];
               if (nodeAt.nodeName=="LI") {
                   nodeAt.onmouseover=function() {
                        this.className ="dropdown";
                    }
                    nodeAt.onmouseout=function() {
                          this.className="";
                    }
             }
      }
}
```

The **for** statement navigates through each childNode inside of elements named **surf**, which are now referenced as **toproot**. Each childNode is assigned to a variable called **nodeAt**. If the childNode is **LI**, the **classNames** are changed according to the mouse movement.

Notice that the style has also been changed:

```
li.dropdown ul { display: block; }
```

This causes list items to be displayed when the className of list items embedded within unordered lists is set equal to dropdown via the for loop in the function.

This code is commonly used to create dropdown menus. To find information on other ways to create dropdown menus, check this out. Have fun and experiment!