JavaScript[*]

R.G. (Dick) Baldwin

Abstract

This module provides an introductory JavaScript programming tutorial that is accessible to blind students with no programming experience.

1 Table of Contents

[*]Version 1.3: Jun 18, 2011 6:28 pm -0500
[†]http://creativecommons.org/licenses/by/3.0/

2 Preface

2.1 General

This module is part of a collection of modules designed to make physics concepts accessible to blind students.

See http://cnx.org/content/col11294/latest/ [1] for the main page of the collection. See http://cnx.org/content/col11294/lat [2] for the table of contents for the collection.

The collection is intended to supplement but not to replace the textbook in an introductory course in high school or college physics.

2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools (as a minimum) to work through the exercises in these modules:

- A graph board for plotting graphs and vector diagrams (http://www.youtube.com/watch?v=c8plj9UsJbg [3]).
- A protractor for measuring angles (http://www.youtube.com/watch?v=v-F06HgiUpw [4]).
- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ [5] .
- A refreshable Braille display capable of providing a line by line tactile output of information displayed on the computer monitor (http://www.userite.com/ecampus/lesson1/tools.php [6]).
- The ability to create tactile graphics as described at http://cnx.org/content/m38546/latest/ [7] .

The minimum prerequisites for understanding the material in these modules include:

- A good understanding of algebra.
- An understanding of the use of a graph board for plotting graphs and vector diagrams (http://www.youtube.com/watch? [8]).
- An understanding of the use of a protractor for measuring angles (http://www.youtube.com/watch?v=v-F06HgiUpw [9]).
- A basic understanding of the use of sine, cosine, and tangent from trigonometry (http://www.clarku.edu/~djoyce/trig/ [10]). (This information will be provided in a later module.)
- An introductory understanding of JavaScript programming (http://www.dickbaldwin.com/tocjscript1.htm [11] and http://www.w3schools.com/js/default.asp [12]). (The purpose of this module is to help you gain that understanding.)
- An understanding of the creation and use of tactile graphics as described at http://cnx.org/content/m38546/latest/ [13] .

[1]http://cnx.org/content/col11294/latest/
[2]http://cnx.org/content/col11294/latest/#cnx_sidebar_column
[3]http://www.youtube.com/watch?v=c8plj9UsJbg
[4]http://www.youtube.com/watch?v=v-F06HgiUpw
[5]http://www.nvda-project.org/
[6]http://www.userite.com/ecampus/lesson1/tools.php
[7]http://cnx.org/content/m38546/latest/
[8]http://www.youtube.com/watch?v=c8plj9UsJbg
[9]http://www.youtube.com/watch?v=v-F06HgiUpw
[10]http://www.clarku.edu/~djoyce/trig/
[11]http://www.dickbaldwin.com/tocjscript1.htm
[12]http://www.w3schools.com/js/default.asp
[13]http://cnx.org/content/m38546/latest/

2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.3.1 Figures

2.3.2 Listings

2.4 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [14] .

3 General background information

I explained in the earlier module titled *Introduction to Accessible Physics Concepts* (see http://cnx.org/content/col11294/late
[15]) why you will need an introductory understanding of JavaScript programming (http://www.dickbaldwin.com/tocjscript1.h
[16] and http://www.w3schools.com/js/default.asp [17]) to understand the modules in this collection. I won't repeat that explanation here. Instead, I will simply launch into the material to help you gain that understanding.

4 Discussion

The goal of this module is to provide an introductory JavaScript programming tutorial that is accessible to blind students with no programming experience.

[14]http://www.dickbaldwin.com/toc.htm
[15]http://cnx.org/content/col11294/latest/
[16]http://www.dickbaldwin.com/tocjscript1.htm
[17]http://www.w3schools.com/js/default.asp

4.1 Why I chose JavaScript

I chose JavaScript for use in these modules for several reasons.

Free

First, JavaScript is free. The capability to program in JavaScript is available to anyone who has a modern browser installed on their computer. Therefore, cost is not an excuse for not learning to program with JavaScript.

If you are reading this module using a modern browser, you have the ability to program using JavaScript immediately. You don't have to go out and buy anything, so that isn't an excuse for putting it off until tomorrow.

If you don't have a modern browser, you can download a free copy of the Firefox browser at http://www.mozilla.com/en-US/firefox/firefox.html[18] .

Fun

Also, programming with JavaScript can be fun. There are a lot of really interesting things that you can do with JavaScript such as playing sound files (see http://www.javascripter.net/faq/sound/play.htm[19]).

OOP

JavaScript encompasses modern programming concepts. For example, JavaScript is based on the concept of objects. Object-Oriented Programming (OOP) is here to stay. (For an extensive discussion of OOP, see the early lessons in my online programming tutorials at http://www.dickbaldwin.com/toc.htm[20] .)

A free audible, tactile scientific calculator

Most importantly, I chose JavaScript because you will be able to use it, along with a screen reader and a Braille display to create your own scientific calculator. You can use JavaScript to create solutions to many of the exercises in the modules in this collection of physics concepts.

4.2 Facilities required

To use JavaScript for its intended purpose in the modules in this collection, you will need the following:

- Access to the Internet, a screen reader, and a Braille display.
- A modern browser such as Firefox 3.6 (see http://www.mozilla.com/en-US/firefox/firefox.html[21]).
- A plain text editor such as Windows notepad, or my favorite, Arachnophilia, which can be downloaded for free at http://www.arachnoid.com/arachnophilia/[22] .

4.3 A minimal JavaScript script

Listing 1 (p. 4) shows a minimal JavaScript script.

Listing 1: A minimal JavaScript script.

```
    <!-- File JavaScript01.html -->
<html><body>
<script language="JavaScript1.3">

document.write("Insert JavaScript between script tags.","</br>")
document.write("Hello from JavaScript")

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

[18]http://www.mozilla.com/en-US/firefox/firefox.html
[19]http://www.javascripter.net/faq/sound/play.htm
[20]http://www.dickbaldwin.com/toc.htm
[21]http://www.mozilla.com/en-US/firefox/firefox.html
[22]http://www.arachnoid.com/arachnophilia/

Run the script

To run this JavaScript script:

- Copy all of the text from the body of Listing 1 (p. 4) into your plain text editor and save the file with an extension of .htm (test.html for example).
- Open that file in your browser. (In most cases, you should be able to simply drag the file and drop it onto an open browser page to open it. If that doesn't work, open it from the browser's **File** menu.)

And the result is...

When you do that, the text shown in Figure 1 (p. 5) should appear in the browser window.

Output from script in Listing 1.

```
    Insert JavaScript between script tags.
Hello from JavaScript
```

Figure 1: Output from script in Listing 1.

That's all there is to it

All you have to do to write and run a JavaScript script is to:

- Copy the text (often referred to as code or source code) from Listing 1 (p. 4) into your plain text editor.
- Replace the code between the two lines containing the word **script** with the code for your new script, leaving the two lines containing the word **script** intact.
- Save the file with an extension of .html. (The file can have any legal file name so long as the extension is .html.)
- Open the new html file in your browser.

When you do that, the script code will be executed.

Display results in the browser window

If your script code contains statements that begin with **document.write** , followed by a pair of matching parentheses, (as shown in Listing 1 (p. 4)), the code in the parentheses will be evaluated and the results of that evaluation will appear in your browser window.

At that point, you will have access to your script code as well as the results of running your script using your screen reader and your Braille display.

4.4 Strings

In programming, we often refer to a group of sequential characters (such as your first name) as a **string** . In JavaScript format (often called syntax), such groups of characters are surrounded by matching quotation marks to cause the group to be recognized as a single string.

The following strings appear in the JavaScript code in Listing 1 (p. 4) :

1. "Insert JavaScript between script tags."
2.

   ```
   "</br>"
   ```

3. "Hello from JavaScript"

The first two strings appear, separated by a comma, inside the matching parentheses following the first occurrence of **document.write** . (We often call the items inside the matching parentheses the argument list.)

The last item in the above list appears in the argument list following the second occurrence of **document.write** in Listing 1 (p. 4) .

4.5 Screen output

If you examine Figure 1 (p. 5) , you will see that the first and third items from the above list (p. 5) appear, without their quotation marks, in the browser window. However, the second item does not appear in the browser window.

Purpose of document.write

Although this is a simplification, for purposes of the modules in this collection, it will suffice to say that the purpose of the **document.write** command is to cause the items in its argument list to be displayed in the browser window. You can place one or more items in the argument list. If there is more than one item in the argument list, the items must be separated by a comma as shown in Listing 1 (p. 4) .

Displaying strings

With some exceptions, items that appear in the argument list surrounded by matching quotation marks (strings) will be displayed in the browser window.

The exceptions include the second item in the above list (p. 5) . This item is a special command to the browser, commonly known as a **break** tag. The occurrence of a **break** tag tells the browser to go down to the next line before displaying any additional material.

Such browser commands usually begin with a left angle bracket as shown in Listing 1 (p. 4) . Because of this, it is usually wise to avoid displaying strings that begin with left angle brackets unless you know for sure that your string won't be interpreted as a command to the browser.

Displaying material other than strings

I will show you how to display material other than strings in a later section titled The string concatenation operator (p. 16) . Before getting into that, however, I will discuss several other topics.

4.6 Structured programming

When a student enrolls in the Object-Oriented Programming course that I teach at my college, I expect that student to have knowledge of something called structured programming, which is generally defined as including the following topics:

1. Functions with parameter passing
2. Sequence
3. Selection (if-else)
4. Repetition (for, while, and do-while loops)

I also expect them to know how to use variables and how to use operators (add, subtract, multiply, divide, etc.).

You will need to have an introductory knowledge of these topics to understand the JavaScript scripts that I will use to explain physics concepts in later modules. I will briefly explain these topics in this module and will discuss them further in later modules where they are used.

4.6.1 Functions

Functions, or procedures as they are called in some languages, provide a fundamental building block for virtually every programming language. The purpose of a function is to encapsulate the ability to perform a task into a single set of code and to be able to execute that code from a variety of different locations within the script. This can eliminate the requirement to repeat the same code over and over when the same task is required at multiple points in the script.

The surface area of a sphere

For example, if you write a script that frequently needs to calculate the surface area of a sphere, you can encapsulate those calculations in a function. Then, whenever your script needs to know the surface area of a sphere, it can simply call the function and provide the radius of the sphere as a parameter. The function will perform the calculation and return the answer to be used by the script at that point.

A JavaScript function definition

A JavaScript function definition has the following basic parts as shown in Listing 2 (p. 7) :

1. The **function** keyword.
2. The function name.
3. A comma-separated list of arguments enclosed in parentheses. (If there are no arguments, which is perfectly legal, the parentheses must still follow the function name but they will be empty.)
4. The statements in the body of the function enclosed in curly brackets.

Two sides to every function

There are two sides to the use of every function:

1. The function definition.
2. The function call.

The definition names the function and specifies how it will behave when it is called.

The call to the function temporarily passes control to the statements in the function causing them to behave as previously defined.

Once the statements have been executed, control is returned to the point in the script where the call was made. The function may, or may not return a value when it returns control.

The argument list

As in the sphere example discussed above, it is often useful to pass information to the function for it to use in doing whatever it is supposed to do (but this is not always required). When we call the function, we include parameters in the call to the function that match up with the argument list mentioned above. That is the mechanism used to pass information to a function. (This will probably make more sense when you see an example. Again, in some cases, no arguments are required.)

The purpose of a function

Usually (but not always), the purpose of a function is to calculate or otherwise determine some value and return it. In the sphere example mentioned earlier, the purpose of the function would be to calculate and return the surface area of the sphere. Returning a value is accomplished using the **return** keyword in the body of the function.

Sometimes, the purpose of a function is not to return a value, but instead to cause some action to occur, such as displaying information in the browser window. In that case, a **return** statement is not required. However, it doesn't cause any problem to put a **return** statement at the end of the function's body with nothing to the right of the word return.

An example function named getHalf

The code in Listing 2 (p. 7) defines a function named **getHalf** and then calls that function from two different locations in a script, passing a different parameter value with each call.

Listing 2: An example function named getHalf.

```
    <!-- File JavaScript02.html -->
<html><body>
<script language="JavaScript1.3">

//This is the syntax for a comment.

//Define the function named getHalf()
function getHalf(incomingParameter) {
  return incomingParameter/2;
}//end function getHalf()

document.write("Call getHalf for 10.6","</br>")
document.write("Half is: ", getHalf(10.6),"</br>");

document.write("Call getHalf again for 12.3","</br>")
document.write("Half is: ", getHalf(12.3));

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

A note about comments

Note the line of code immediately following the first **script** tag that begins with //. Whenever JavaScript code contains such a pair of slash marks (that are not inside of a quoted string), everything from that point to the end of the line is treated as a comment. A comment is intended for human consumption only and is completely ignored when the script is run.

The function definition

The function definition in Listing 1 (p. 4) consists of the three lines of code following the comment that begins with "//Define the function..."

As explained earlier, this function definition contains:

- The **function** keyword
- The function name: **getHalf**
- An argument list containing a single parameter named **incomingParameter**
- A function body enclosed in a pair of matching curly brackets

Simpler than the surface area of a sphere

The function named **getHalf** is somewhat simpler than one that could be used to calculate the surface area of a sphere, but the basic concept is the same. This function expects to receive one parameter.

The code in the body of the function uses the division operator, "/", to divide the value of the incoming parameter by 2. Then it returns the result of that calculation. When the function returns, the return value will replace the call to the function in the calling script.

Two calls to the function

The function is called twice in the body of the script in Listing 2 (p. 7) , passing a different value for the parameter during each call.

Each call to the function named **getHalf** is embedded as one of the elements in the argument list following **document.write** .

write is also a function

Although I didn't mention this earlier because you weren't ready for it yet, **write** is also the name of a function. However, the **write** function is predefined in JavaScript and we can use it to display information in the browser window without having to define it first.

(Actually, **write** is a special kind of a function that we call a **method** , but you don't need to worry about that unless you want to dig much deeper into the object-oriented aspects of JavaScript.)

Two calls to the getHalf function

The script in Listing 2 (p. 7) calls the function named **getHalf** twice in two different locations, each of which is embedded in the argument list of a call to the **write** method. Each call passes a different parameter value to the **getHalf** function.

The order of operations

When a call to a function or method (such as the call to the **write** method) includes a call to another function or method in its argument list, the call to the function in the argument list must be completed before the call can be made to the function having the argument list. Therefore in each of these two cases, the call to the **getHalf** function must be completed before the call to the **write** method can be executed.

Output from script in Listing 2 (p. 7)

Each call to the **getHalf** function returns a value that is half the value of its incoming parameter.

As I mentioned earlier, when the function returns, the returned value replaces the call to the function. Therefore, in each case, the returned value from the call to the **getHalf** function becomes part of the argument list for the **write** method before that method is called. This causes the two calls to the **write** method in Listing 2 (p. 7) to display the values returned from the calls to the **getHalf** function as shown in Figure 2 (p. 9) .

Output from script in Listing 2.

```
    Call getHalf for 10.6
Half is: 5.3
Call getHalf again for 12.3
Half is: 6.15
```

Figure 2: Output from script in Listing 2.

Good programming design

It is a principle of good programming design that each function should perform only one task, and should perform it well. The task performed by the function named **getHalf** is to calculate and return half the value that it receives, and it does that task very well.

4.6.2 Arithmetic operators

The division operation in Listing 2 (p. 7) introduced the use of arithmetic operators.

In computer programming jargon, we speak of operators and operands. Operators operate on operands.

As a real-world example, if you were to go to the hospital for knee surgery, the surgeon would be the **operator** and you would be the **operand** . The surgeon would operate on you.

Binary operators

The operators that we will use in the modules in this collection will usually be restricted to those that have two operands, a **left operand** and a **right operand** . (Operators with two operands are commonly called **binary operators** .)

In the function named **getHalf** in Listing 2 (p. 7) , the "/" character is the division operator. The left operand is **incomingParameter** and the right operand is **2** . The result is that the incoming parameter is divided by the right operand (2).

Binary arithmetic operators

The binary arithmetic operators supported by JavaScript are shown in Figure 3 (p. 10) .

Binary arithmetic operators.

```
+ Addition:        Adds the operands
- Subtraction:     Subtracts the right operand from the left operand
* Multiplication:  Multiplies the operands
/ Division:        Divides the left operand by the right operand
% Modulus:         Returns integer remainder of dividing the left operand
                   by the right operand
```

Figure 3: Binary arithmetic operators.

We will use these operators extensively as we work through the physics exercises in future modules.

4.6.3 Sequence

Of the four items listed under Structured programming (p. 6) earlier, the simplest one is **sequence** .

The concept of sequence in structured programming simply means that code statements can be executed in sequential order. Listing 2 (p. 7) provides a good example of the sequential execution of statements. The code in Listing 2 (p. 7) shows four sequential statements that begin with **document.write.**

4.6.4 Selection

The next item that we will discuss from the list under Structured programming (p. 6) is **selection** . While not as simple as **sequence** , selection is something that you probably do many times each day without even thinking about it. Therefore, once you understand it, it isn't complicated.

General syntax for selection statement

The general syntax for a selection statement (often called an **if-else** statement) is shown in Figure 4 (p. 11) .

General syntax for selection statement.

```
if(conditional expression is true){
  execute code
}else{//optional
  execute alternative code
}//end selection statement
```

Figure 4: General syntax for selection statement.

A selection statement performs a logical test that returns either true or false. Depending on the result, specific code is executed to control the behavior of the script.

A real-world analogy

Figure 5 (p. 11) shows a real-world analogy of a selection statement that you might make on your day off.

Real-world analogy of a selection statement.

```
if(it is not raining){
  Play tennis
  Go for a walk
  Relax on the beach
}else{//optional
  Make popcorn
  Watch TV
}//end selection statement
```

Figure 5: Real-world analogy of a selection statement.

In this analogy, you would check outside to confirm that it is not raining. If the condition is true (meaning that it isn't raining), you would play tennis, go for a walk, and then relax on the beach. If it is raining, (meaning that the test condition is false), you would make some popcorn and relax in front of the TV.

The else clause is optional

Note that when writing JavaScript code, the **else** clause shown in Figure 5 (p. 11) is optional. In other words, you might choose to direct the script to take some specific action if the condition is true, but simply transfer control to the next sequential statement in the script if the condition is false.

A selection script example

Listing 3 (p. 12) shows a sample script containing two selection statements (commonly called **if-else** statements) in sequence.

Listing 3: A selection script example.

```
    <!-- File JavaScript03.html -->
<html><body>
<script language="JavaScript1.3">

if(3 > 2){
document.write("3 is greater than 2.","</br>")
}else{
document.write("3 is not greater than 2.")
}//end if

if(3 < 2){
document.write("3 is less than 2.","</br>")
}else{
document.write("3 is not less than 2.")
}//end if

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

If 3 is greater than 2...

The conditional clause in the first **if** statement in Listing 3 uses the "greater-than" relational operator ">" to determine if the literal value 3 is greater than the literal value 2, producing the first line of output shown in Figure 6 (p. 12) .

Output from script in Listing 3 .

```
    3 is greater than 2.
3 is not less than 2.
```

Figure 6: Output from script in Listing 3 .

The test returns true

Since 3 is always greater than 2, the statement in the line immediately following the first test in Listing 3 (p. 12) is executed and the code in the line following the word **else** is skipped producing the first line of output text shown in Figure 6 (p. 12) .

If 3 is less than 2...

The conditional clause in the second **if** statement in Listing 3 (p. 12) uses the "less-than" relational operator (see Figure 7 (p. 13)) to determine if the literal value 3 is less than the literal value 2.

The test returns false

Since 3 isn't less than 2, the statement in the line immediately following the second test in Listing 3 (p. 12) is skipped and the statement in the line immediately following the second word **else** is executed producing the second line of output text shown in Figure 6 (p. 12) .

4.6.5 Relational and logical operators

I doubt that I will need to use logical operators in the modules in this collection. If I do, I will explain them at the time. However, I will use relational operators.

The relational operators that are supported by JavaScript are shown in Figure 7 (p. 13) .

Relational operators.

```
>  Left operand is greater than right operand
>= Left operand is greater than or equal to right operand
<  Left operand is less than right operand
<= Left operand is less than or equal to right operand
== Left operand is equal to right operand
!= Left operand is not equal to right operand
```

Figure 7: Relational operators.

As with the arithmetic operators discussed earlier, we will use these operators extensively as we work through the physics exercises in future modules.

4.6.6 Variables

The next item in the list under Structured programming (p. 6) is **repetition** . Before I can explain repetition, however, I need to explain the use of variables.

What is a variable?

You can think of a variable as the symbolic name for a pigeonhole in memory where the script can store a value. The script can change the values stored in that pigeonhole during the execution of the script. Once a value has been stored in a variable, that value can be accessed by calling out the name of the variable.

Variable names

Variable names must conform to the naming rules for identifiers. A JavaScript identifier must start with a letter or underscore "_". Following this, you can use either letters or the symbols for the digits (0-9) in the variable name.

JavaScript is case sensitive. Therefore letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Declaring a variable

In many languages, including JavaScript, you must *declare* a variable before you can use it. However, JavaScript is very loose in this regard. There are two ways to declare a variable in JavaScript:

- By simply assigning it a value; for example, x = 10
- By using the keyword **var** ; for example, **var** x = 10

Scope

When working with variables in JavaScript and other languages as well, you must always be concerned about an issue known as scope. Among other things, scope determines which statements in a script have access to a variable.

Two kinds of variables

JavaScript recognizes two kinds of variables:

- local
- global

Local variables are variables that are declared inside a function. Global variables are variables that are declared outside a function.

Scope

Local variables may only be accessed by other code within the same function following the declaration of the variable. Hence, the scope of local variables is limited to the function or method in which it is declared.

Global variables may be accessed by any code within the script following the declaration of the variable. Hence, the scope of global variables is the entire script.

Use of the keyword var

Using **var** to declare a global variable is optional. However, you must use **var** to declare a variable inside a function (a local variable).

A sample script that uses variables

A sample script that uses variables to store data is shown in Listing 4 (p. 14) .

Listing 4: A sample script that uses variables.

```
  <!-- File JavaScript04.html -->
<html><body>
<script language="JavaScript1.3">

//Define the function named getArea()
function getArea(theRadius) {
  //declare a local variable
  var theArea

  //calculate the area
  theArea = Math.PI * theRadius * theRadius
  return theArea
}//end function getArea()
//========================================//
```

```
//declare a global variable without keyword var
area = getArea(3.2)//call the function
document.write("Area is: " + area)

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

The area of a circle

Listing 4 (p. 14) begins by defining a function that will compute and return the area of a circle given the radius of the circle as an incoming parameter.

The code in the function begins by declaring a variable named **theArea** . Effectively, this declaration sets aside a pigeon hole in memory and gives it the name **theArea** . Once the variable is declared, it can be accessed by calling out its name. It can be used to store data, or it can be used to retrieve data previously stored there.

Behavior of the function named getArea

You may recall that the area of a circle is calculated by multiplying the mathematical constant PI by the radius squared. There is no squaring operator in JavaScript. Therefore, you can square a value by multiplying it by itself.

PI times the square of the radius

The code in the function named **getArea** in Listing 4 (p. 14) computes the area and uses the **assignment** operator "=" to store the result in the variable named **theArea** . This statement represents a case where a value is being stored in a variable (assigned to the variable) for safekeeping.

Then the code in the function extracts the value stored in the variable named **theArea** and returns that value. After that, the function terminates and returns control to the place in the calling script from which it was originally called.

The variable named area

Further down the page in Listing 4 (p. 14) , the script declares a variable named **area** without using the keyword **var** . (Note, however, that the keyword **var** could have been used here. I prefer that approach for reasons that I won't get into here.)

The script calls the **getArea** function, passing a radius value of 3.2 as a parameter. As you learned earlier, the value returned by the function replaces the call to the function, which is then assigned to the variable named **area** .

Display the results

Then the script calls the **write** method to display some text followed by the value stored in the variable named **area** , producing the output shown in Figure 8 (p. 15) in the browser window.

Output from script in Listing 4 .

```
Area is: 32.169908772759484
```

Figure 8: Output from script in Listing 4 .

4.6.7 The string concatenation operator

The code in Listing 4 (p. 14) exposes another operator that I will refer to as the *string concatenation operator* .

Note the argument list for the call to the **write** method in Listing 4 (p. 14) . In addition to being used to perform numeric addition, the plus operator "+" can be used to concatenate (join) two strings.

If two strings are joined by the + operator, the two strings will produce a new string that replaces the combination of the two original strings and the + operator.

If the left operand to the + operator is a string and the right operand is a numeric value (or the name of a variable containing a numeric value), the numeric value will be replaced by a string of characters that represent that numeric value and the two strings will be concatenated into a single string.

4.6.8 Repetition

The last item in the list under Structured programming (p. 6) is **repetition** , and that will be the topic of this section.

Repetition (also referred to as looping) means the act of causing something to repeat.

A repetition or loop is a set of commands that executes repeatedly until a specified condition is met.

JavaScript supports three loop statements:

- while
- for
- do-while

The while loop

The **while** loop is not only the simplest of the three, it is also the most fundamental. It can be used to satisfy any requirement for repetition in a JavaScript script. The other two exist solely for added convenience in some situations. Therefore, I will concentrate on the **while** loop, and leave the other two be be discussed in a future module, if at all.

Loop while a condition is true

A **while** loop executes the statements in its body for as long as a specified condition evaluates to true. The general syntax of a **while** loop is shown in Figure 9 (p. 16) .

General syntax for a while loop.

```
while(condition is true){
  //Execute statements in body of loop.
}//end while statement
```

Figure 9: General syntax for a while loop.

When the condition is no longer true...

When the conditional expression evaluates to false, control passes to the next statement following the **while** loop.

while loops can be nested inside of other statements, including other **while** loops.

An infinite loop

As with all loop statements, you must be careful to make certain that the conditional expression eventually evaluates to false. Otherwise, control will be trapped inside the **while** loop in what is commonly called an infinite loop.

A simple while loop

The script in Listing 5 (p. 17) illustrates the use of a simple **while** loop.

Listing 5: A simple while loop.

```
    <!-- File JavaScript05.html -->
<html><body>
<script language="JavaScript1.3">

cnt = 4//initialize a counter variable
while(cnt >= 0){//begin while loop
  //display value of counter
  document.write("cnt = " + cnt + "</br>")
  cnt = cnt - 1//decrement counter
}//end while loop

document.write("The End.");

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

A counting loop

This script initializes the value of a counter variable named **cnt** to 4. Control continues to loop (iterate) for as long as the value of the counter is greater than or equal to zero. During each iteration of the loop, the current value of **cnt** is displayed and then the value of **cnt** is reduced by a value of 1.

Repeat the test

At that point, control returns to the top of the loop where the test is repeated. This process produces the first five lines of output text shown in Figure 10 (p. 18) .

Output from script in Listing 5 .

```
    cnt = 4
cnt = 3
cnt = 2
cnt = 1
cnt = 0
The End.
```

Figure 10: Output from script in Listing 5 .

When the test returns false...

When the test in Listing 5 (p. 17) returns false, meaning that **cnt** is no longer greater than or equal to zero, control exits the **while** loop and goes to the next statement following the **while** loop. This is the statement that calls the **write** method to display the last line of text in Figure 10 (p. 18) .

4.7 Programming errors

From time to time, we all make errors when writing scripts or programs. Typical errors include typing a period instead of a comma, failing to include a matching right parenthesis, etc. Usually, when you make a programming error using JavaScript, some or all of the script simply doesn't execute.

Finding and fixing the errors

The worst thing about programming errors is the need to find and fix them. The Firefox and Google Chrome browsers have easy-to-use mechanisms to help you identify the cause of the problem so that you can fix it. Internet Explorer probably has similar capability, but so far, I haven't figured out how to access it.

My recommendation is to simply open the files containing your JavaScript code in either Firefox or Google Chrome. Or, you can open the file in Internet Explorer, but if it doesn't do what you expect it to do, open it again in Google Chrome or Firefox for assistance in finding and fixing the problem.

4.7.1 Assistance using Google Chrome

You can open the JavaScript console in the Chrome browser by holding down the Ctrl key and the Shift key and pressing the J key. The console will open at the bottom of the Chrome browser window. You can also close the console with the same keystroke.

The format of the console is a little messy and may be difficult for blind students to navigate. However, it can be useful in locating errors if you can navigate it.

An error message in the console

If you open an html file containing a JavaScript error in the browser while the console is open, an error message will appear in the console. For example, I am looking at such an error as I type this document. It consists of a round red circle with a white x followed by the following text:

"Uncaught SyntaxError: Unexpected number"

The file name and line number

On the far right side of the same line is text that reads junk.html:23. That is the name of the file and the line number in that file containing the error. That text is a hyperlink. If the hyperlink is selected, another part of the console opens showing the offending line of JavaScript code. I'm not sure that would be useful to a blind student, because navigation within the console would probably be a problem.

The description is unreliable

Also, in this particular case the description of the error isn't very useful in determining the cause of the error although sometimes it may be useful. My advice is not to put too much faith in that description. The error was actually a missing relational operator in a comparison clause.

The line number is very important

Probably the most useful information is the line number that you can use to go back and examine your source code, looking for an error in that line of code.

4.7.2 Assistance using Firefox

You can open an error console when using the Firefox browser by holding down the Ctrl key and the Shift key and pressing the J key. The console will open in a separate window. Unlike with Chrome, repeating the keystroke won't close the error console.

An error message in the console

If you open an html file containing a JavaScript error in the browser while the error console is open, an error message will appear in the console. For example, I am looking at such an error as I type this document. It consists of a round red circle with a white x and the following text:

missing) after condition

file: –html file name and path here– Line: 23

while(h 0){

The middle line is a hyperlink

The middle line of text that contains the file name to the left of the line number is a hyperlink. In this case, the hyperlink may be useful, depending on how things are treated by your screen reader and your Braille display. If you select the link, a window will open showing the source code with the problem line highlighted. Pressing the right arrow key will cause a blinking cursor to appear between the first and second characters in that line. If a blind student can identify the highlighted line with the blinking cursor, that would be useful.

The description is unreliable

As with Chrome, in this particular case the description of the error isn't very useful in determining the cause of the error although sometimes it may be useful. My advice is not to put too much faith in that description. The error was actually a missing relational operator in a comparison clause.

The line number is very important

Probably the most useful information is the line number that you can use to go back and examine your source code, looking for an error in that line of code.

5 Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of html. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6 Resources

I will publish a module containing consolidated links to resources on my Connexions web page and will update and add to the list as additional modules in this collection are published.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: JavaScript
- File: Phy1010.htm
- Revised 06/18/2011
- Keywords:
  - physics
  - accessible
  - blind
  - graph board
  - protractor
  - screen reader
  - refreshable Braille display
  - JavaScript
  - trigonometry
  - function
  - argument
  - parameter
  - sequence
  - selection
  - repetition
  - structured programming
  - string
  - arithmetic operators
  - relational operators
  - string concatenation
  - variable

-end-