JavaScript

Here are notes about JavaScript. They're intended primarily to introduce the idea of variables, functions, data types, and the integration of html (for presentation) with a scripting language (for dynamism) using JavaScript.

[We will replace JavaScript with alternatives, such as server-side processing [.jsp, .asp, servlets, php], or combinations using Java applets.]

JavaScript is a programming language that whose commands reside in a web page itself or are linked to the web page in a separate file, e.g,: `<script type="text/javascript" src="browserCheck.js"></script>`. In the former case, the HTML page includes the following tag to indicate to the browser the start of something that is not html:

```
<script type="text/javascript">
 (place script here)
</script>
```

The `<script>` tag causes the browser not to interpret the words between `<script>` and `</script>` as html but rather as something different - in this case the browser is instructed to read the words as "text" and interpreted them using javascript.
The MIME type is `text/javascript`.

Browsers have been pre-programmed to interpret HTML tags and JavaScript. It's up to the web page designer to let the browser know what to do. In these examples only the basic HTML tags are used.

In this section about JavaScript, the fundamentals should be mastered by all LIS489 students. Students of other classes should master all the issues in this section.

Example:

```
<html>
<head>
<title>A basic web page with internal java script</title>
     <script type="text/javascript">
     <!-- this is a comment - use them liberally in your work. -->
          // This is a java script comment for a single line. //
          /* this comment is for multiple lines.
          In this demo use call the alert() function.  This is one
          of the commands built-in to JavaScript.
          */
          alert("Welcome to JavaScript");
     </script>
</head>
<body>This is a demo html with a javascript.</body>
</html>
```

What to know
Script MIME type (see the MIME readings for LIS489.]
JavaScript is built-into the browser; we call the commands through the browser.
Scripts that appear before the `</head>` tag can be used any place in the rest of the web page. [They have "global scope."]

Variables:

A variable acts like a bucket to hold something. In programming languages, a variable needs a name and to be assigned a value. For example, we have a word "age" that we want to have a value "25". We would write "age = 25". This means there's a location in the computer's memory that holds the value "25". We access this value by using the variable name, "age". We must indicate to JavaScript that we're using a variable (and not just typing words) by using a control word "var". As you will see shortly, inside the area indicated by the <script> tag will hold our variables. It will hold also commands that we write, called functions. In tis example, we are going to create a var: var age = 25;

Example:

```
<script type="language="text/javascript">
     var age = 25;
</script>
```

Notice the 5 parts:

- "var" means "create a bucket in memory to hold something … "
- age is the name of the variable;
- = is the assignment; the use of this equal sign = tells JavaScript to map the variable "age" to the value 25 (in other words, store the 25 in the bucket called "age")
- 25 - notice that there are no quote marks. No quotes tells JavaScript that 25 is a number (actually an integer, or counting number). We can perform arithmetic on this value.
- ; Notice that the end of the command must be indicate, too. The semi-colon tells the programming language when to stop reading a command.

If we said `age = '25'` it means the 25 is not a number but a string of two characters (that happen to look like numbers to us).

Compare: `var resultOfAddition = 34 + 12;` The actual value in our bucket would be `46`. If we said `var resultOfAddition = "cat" + "fish";` we would set the value to "`catfish`".

Operators: *There are are number of operators to recognize:*

```
=    equals
+    add
-    subtract
*    multiply
/    divide
+=   add 1 to previous value (increment), e.g., var age += 25 means 25 +
1
-=   subtract 1 from previous value (decrement), e.g., var age -= 25; 25
- 1 = 24.
*=   same as above but with multiplication
/=   same but with division
==   equivalency test
```

(to see if two variables are equal, e.g., if myVar == "cat" tests if the value in myVar is the same as the string "cat"; e.g., if (age == 25).

Statements

The "var" above is actually a "statement" - it declares something to the programming language.

Conditionals:

An "expression" is a statement that the programming language must interpret and decide how to proceed.

For example, if a student's quiz score is > 90 percent, assignment the grade of A. We express this idea through "conditional statements." Example: If the score is > 90, assignment an A, else assign a B.

There are several varieties that you should know.

Template *Example*

if - one statement

```
if (expression) {
     statement;
}
```

```
if (myvar == 25) {
          salary *= 1.25;
}
```

if - more than one statement

```
if (expression) {
     statement1;
} else {
     statement2;
}
```

```
if (myvar == 25) {
     salary *= 1.25;
} else {
     salary *= 2.00;
}
```

Example

```
<html><head><title>demo</title>
<script type="text/javascript">
   var a = 2;
   var b = 3;
   if (a == 0) {
      document.write("The variable is 0");
   } else {
      /* NOTICE WE HAVE AN if STATEMENT NESTED IN ANOTHER IF STATEMENT */
      if ((a * 2) >= b) {
         document.write("a * 2 is greater than b");
      } else {
         document.write("a* 2 is less than b");
      }
   }
</script>
</html>
```

Other Conditionals:

`If … then … else`

When there are multiple choices you can use multiple if statements.

```
<html>
<head>
<title>if...then...else...if</title>
<script type="text/javascript">
var stateCode = 'MO';

if (stateCode == 'OR') {
   taxPercentage = 3.5;
} else if (stateCode == 'CA') {
   taxPercentage = 5.0;
} else if (stateCode == 'MO') {
   taxPercentage = 1.0;
} else {
```

```
   taxPercentage = 2.0;
}

document.write(taxPercentage);

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

Example: using if/else to check the user's input:

```
<html>
<head>
<title>A Choice Page</title>
<script type="text/javascript">
<!--
var response = confirm("Do you want to proceed with this book?  Click OK to proceed otherwise click Cancel.");
if ( response == true )
{
   alert("A fine choice!")
}else{
   alert("A not fine choice!")
}
//  -->
</script>
</head>
<body></html>
```

For Loops These are exceptionally useful.

- The for loop is a structure that loops for a preset number of times.
- The for loop is made up of two parts: condition and statement.
- The condition structure determines how many times the loop repeats.
- The statement is what is executed every time the loop occurs.
- The condition structure is contained within parentheses and is made up of three parts.
- Each part is separated by a semicolon (;).
- The first part of the condition structure initializes a variable to a starting value.
- In most cases, the variable is declared within this section as well as initialized.
- The second part is the conditional statement.
- The third and final part determines how the variable should be changed each time the loop is iterated.

The format of the for loop looks like the following:

```
for (initialize; condition; adjust)  {
      statement;
}
```

Example:

```
<html><head><title>A demo</title>
<script type="text/javascript">
    <!--
    document.write("<H2>Multiplication table for 4</H2>");

    for (var aNum = 0; aNum <= 10; aNum++)
    {
```

```
      document.write("4 X ",aNum," = ",4*aNum,"<BR>");
    }
    //-->
</script>
</head>
<body> </body>
</html>
```

In this example, we combine what we've seen so far - variables, a for example, post-fix operator (i++), etc.

```
<html>
<head>
<title>using the javascript for loop
</title>
</head>
<body>
<h1>using the javascript for loop</h1>
<script type="text/javascript">
<!--
var i, total
total = 0
for (i = 1; i < 5; i++) {
   total += i
   document.write("loop iteration " + i + " (cumulative total = "  + total + ")<
br>")
}
// -->
</script>
</body>
</html>
```

Switch

JavaScript offers the switch statement as an alternative to using the if...else structure.
The switch statement is useful when testing more than a few if… possibilities and makes clearer what to do in a default situation. Switches are also processed more efficiently by the computer. Here's the format of a switch statement:

```
switch (expression)
    {
      case label1:
        statement1;
        break;
      case label2:
        statement2;
        break;
      default:
        statement3;
    }
```

The switch statement evaluates an expression placed between parentheses.
The result is compared to labels associated with case structures that follow the switch statement.
If the result is equal to a label, the statement(s) in the corresponding case structure are executed.
A default is used at the end of a switch structure to catch results that do not match any of the case labels.
A colon always follows a label.

Curly brackets {} are used to hold all the case structures together, but they are not used within a case structure.
The keyword `break` is used to break out of the entire switch statement once a match is found, thus preventing the default structure from being executed accidentally.

```
<html>
<script type="text/javascript">
<!--
    var color = "green";
    switch (color)
    {
      case "red":
        document.write("The car is red.");
        break;
      case "blue":
        document.write("The car is blue.");
        break;
      case "green":
        document.write("The car is green.");
        break;
      default:
        document.write("The car is purple.");
    }
    //-->
</script>
</html>
```

Example
In this example we use a `form` tag to accept input from the user and then send the input to JavaScript. The function "`verifyDay()`" accepts this input as a `parameter`. Depending on the value of the parameters, the function's switch statement responds differently.

```
<html>
    <head>
      <title>Using the switch statement</title>
    <script type="text/javascript">
    <!--
    function verifyDay(form){
      var myEntry = form.day.value.toUpperCase();

      var firstPart = "You have entered a day at the first of the week";

      var endPart = "You have entered a day at the end of the week";

      var weekEnd = "You have entered a weekend day";

      switch(myEntry){
        case "MONDAY" :
          alert(firstPart);
          break;
        case "TUESDAY" :
          alert(firstPart);
          break;
        case "WEDNESDAY" :
          alert('You ave entered a "hump" day');
          break;
```

```
        case "THURSDAY" :
          alert(endPart);
          break;
        case "FRIDAY" :
          alert(endPart);
          break;
        case "SATURDAY" :
          alert(weekEnd);
           break;
        case "SUNDAY" :
          alert(weekEnd);
          break;
        default :
          alert("Sorry, that's an invalid day");
      }
    }
    -->
    </script>
    </head>
    <body>
    <form name="myForm">
    <b>Please enter a day of the week:</b><br>
      <input type=text value="" name="day">
      <input type=button value="Verify" name="myButton"
            onClick='verifyDay(this.form)'>
    </form>
    </body>
</html>
```

Sometimes we want to make sure we know what the user input. Once we have the value, we can use the data for other work. In this example, the user's input is stored in the value "yourchoice".

Note that the variable yourchoice is declared but not initiated. [That means the program knows about a variable but has not yet assigned a value - it's kind of an empty bucket waiting to be filled.] … cut from webpage ...

```
<html>
<head>
<title>A Simple Page</title>
<script type="text/javascript">
<!--
var yourchoice;
yourchoice = prompt("Choose a number between 1 and 4", "1, 2, 3 or 4")
switch (yourchoice)
{
    case "1":
        alert("You typed in a 1");
        break;
    case "2":
        alert("You typed in a 2");
        break;
    case "3":
        alert("You typed in a 3");
```

```
        break;
    case "4":
        alert("You typed in a 4");
        break;
}
//  -->
</script>
</head>
<body>
...
</body>
</html>
```

Default values...

There are times when you want to make sure the user has entered a value that is appropriate. This is called "error checking on input." Making sure the user's input is valid or isn't empty (or null) before the data are sent to the web server will save time and trouble and materially affect the programming of server applications. [Otherwise the server application would have to check for null and inappropriate values and then send error messages back to the user.]

```
<html>
<head>
<title>A Simple Page</title>
<script type="text/javascript">
<!--
var yourchoice;
yourchoice = prompt("Choose a number between 1 and 4", "1, 2, 3 or 4")
switch (yourchoice)
{
    case "1":
        alert("You typed in a 1");
        break;
    case "2":
        alert("You typed in a 2");
        break;
    case "3":
        alert("You typed in a 3");
        break;
    case "4":
        alert("You typed in a 4");
        break;
}
//  -->
</script>
</head>
<body> … </body>
</html>
```

Booleans

Besides integers and Strings (alphanumerics, a-z, A-Z, 0-9 - pretty much anything you see on

the keyboard), there are "booleans". These are "true", "false". In this example we rely on the pre-programming of the `alert()` function. Try typing and running this script.

```
<html>
<head>
<title>A Simple Page</title>
<script type="text/javascript">
<!--
var yourchoice;
yourchoice = confirm("Are you at your computer now?")
switch (yourchoice)
{
    case true:
        alert("true");
        break;
    case false:
        alert("false");
        break;
}
//  -->
</script>
</head>
<body>

</body>
</html>
```

Loops and Counters

Sometimes you know you want something to happen for a number of times. Say you want to ring the doorbell 5 times. You might write a function called "ringTheBell()" that [somehow] automatically rings your front door bell. (We haven't really written this function; just pretend for the moment that we have a function that will actually ring your doorbell the number of times between the ( ). [These are the parameters, remember?]
You need a variable to hold the number of times the bell has already been run. The program uses this value also to know when to stop. So, if we haven't wrung the bell yet, we might declare a variable called "noOfTimesWrung" and set it to equal 0: var noOfTimesWrung = 0;

In this example, notice we use a "while" loop.

```
<html>
<head>
<title>Ring the Bell! Page</title>
<script type="text/javascript">
<!--
var noOfTimesWrung = 0;
while (noOfTimesWrung < 5)
{
    ++noOfTimesWrung;
    ringTheBell(noOfTimesWrung);
}
```

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

</body>
</html>
```

A version of the **while** loop is a **do … while** loop. Compare the structure of this loop to the previous one:

```
<html>
   <body>
   <h1>
   <script>
      var newstring = "";
      var thestring = 'abcde';
      var counter = thestring.length;
      do
      {
         newstring += thestring.substring(counter-1, counter);
         counter--;
      }
      while (counter > 0 );
      document.write(thestring + " reversed is " + newstring + "!");
   </script>
   </h1>
   </body>
</html>
```

Modula

In this example, notice two new things: the modula operator (%) that's very useful for automatically changing the color of alternating lines and notice the "continue;" statement. This is a way of skipping over certain situations. In this case, only if the value of x when divided by 2 = 0 (that is, only when x is even) skip it; when x is odd, then write the value of x.

```
<html>
<head>
<title>A Simple Page</title>
<script language="JavaScript">
<!--
var x = 0;
while (x < 10)
{
    x++;
    if (x % 2 == 0)
    {
        continue;
    }
```

```
    document.write(x);
}
//  -->
</script>
</head>
<body>

</body>
</html>
```

Above we wanted to *continue* ; there are times when you want to jump out of something - or "break".

```
<html>
    <script type="text/javascript">
    <!--
    forloop1:
    for (var counter1 = 1; counter1 <= 5; counter1++)
    {
      for (var counter2 = 1; counter2 <= 5; counter2++)
      {
        document.write("counter1=",counter1);
        document.write(" counter2=",counter2,"<br>");
        if (counter2 == 3)
          break;
        if (counter1 == 3)
          break forloop1;
      }
    }
    document.write("all done!");
    //-->
    </script>
</html>
```

Here is another example, now combining a couple of conditionals:

```
<html>
<head>
<title>A Simple Page</title>
<script type="text/javascript">
<!--
var x = 0, breakat;
breakat = prompt("Pick a number between 1 and 10 to break at", "");
while (x < 10) {
    if (x == breakat)
        break;
    }
    document.write(x);
    x++;
}
//  -->
</script>
```

```
</head>
<body>

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

Many times in programming (and life) you want to make sure that *all* activities are completed before saying the job is done. For example, say you have a task that requires 3 steps to be completed before you can declare the job finished. In computing, there are many steps that the end-user doesn't see but we have to make sure about. For instance, say you want to save data from a web form into a relational database. We must make sure first that we can get a connection to the database, that we have permission to write the data to the database table, and that we're successful at storing the data in the file. Then we have to make sure we close the connection to the database. Each of these steps can succeed or they can fail. If one part fails, then the entire process should not be judged complete and correct.

There is a block of code called a "try … catch" block. This means "try to do everything required and if it doesn't work, catch whatever error is created (or "thrown") and undo anything that's been attempted so far.

Some programming activities, such as communicating with a database or reading/writing a file *require* a try… catch block. Sometimes, in the programming language documentation, you will read the phrase "throws Exception". This means whatever function you're using must be wrapped in a try … catch block.

Here's the syntax:

```
try {
        statement1
}
catch(exception) {
        statement2
}
```

- It can be used to handle all or some of the errors that can occur in a script.
- If an error is not handled by a try...catch statement, it is passed on so other statements can handle the error.
- If there are no other statements to handle the error, it is passed to the browser to handle.
- statement1 is where an error can occur, while statement2 is used to handle the error.
- As soon as an error occurs, the value thrown is passed to the catch portion of the statement and stored in exception.
- If the error can not be handled, another throw statement is used to pass the error to a higher level handler if one is defined.
- It is possible to have nested try...catch statements within try...catch statements.

The following example uses a try...catch Statement to Handle an Incorrect Entry.

```
<html>
    <head>
      <title>Using a try..catch statement</title>
      <script type="text/javascript">
    <!--
    function myErrorHandler(data){
      try{
        try{
          if(data == "string"){
            throw "E0";
          }else{
            throw "E1";
          }
        }catch(e){
          if(e == "E0"){
            return("Error (" + e + "): Entry must be numeric.");
          }else{
            throw e;
          }
        }
      }catch (e){
        return("Error (" + e + "): Entry was invalid.");
      }
    }
    function processData(form){
      if(isNaN(parseInt(form.myText.value))){
        alert(myErrorHandler("string"));
      }else{
        alert("You have correctly entered a number");
      }
    }
    -->
    </script>
    </head>
    <body>
    <form name="myForm">
      Please enter a number:
      <input type=text size=10 value="" name="myText">
      <input type=button value="Process" name="myButton"
            onClick='processData(this.form)'>
    </form>
    </body>
 </html>
```

In the above example note several important issues.

Errors:

Notice in the "catch()" the letter "e". Whenever a program runs, it can keep track of its own behaviors. There are a number of built-in functions that the end-user never sees. One of those built-in functions handles errors. So, when an error occurs, the program creates an "error object" and stores the error number and message. We can extract the error message and display it. For

example, if you want to read a file and the file is not on the disk, you might get an error message "File not found."

In this example, the "error object" that is created is automatically given a name: in this case "e". We, the programmers, decided to call it that. [We could call it "z" if we felt like it.] What matters is our catch statement is ready for a parameters to be passed to it - the error object that JavaScript created when there's an error. We are specifically asking JavaScript to give us a copy of the error message (that we've called "e") and now we're going to display that error on the screen.

These errors are called **exceptions**. This is also another example of abstraction in computers. Instead of saying "error", we call it an "exception" - perhaps something useful actually happened but it wasn't what we wanted at that time. It's not an error, just not what we need. So, in general, when something doesn't quite go the way we want (error) or when the program's providing more data about itself than we need, such as a "status message"), we call them "Exceptions" instead of errors.

In this example, an exception is thrown and we present the message to the end-user, via a JavaScript alert() function.

```
<html>
<head>
<title></title>

<script type="text/javascript">
function getmonthname (monthnumber) {
     throw "invalidmonthnumber"
}
try {
    alert(getmonthname(13))
}
catch (exception) {
    alert("an " + exception + " exception was encountered.
      please contact the program vendor.")
}
</script>
<body>

</body>
</html>
```

Note this example: Exceptions are generated by various technologies - some are created by database management systems, some by the program itself, some by the operating system! In this example, we look for specifically an error that was generated by a problem in the programming (the syntax). Here we ask the "generic exception" to give us only a SyntaxError. In other words we look for an "instanceof" a particular set of errors. Here we ask for errors in our scripting:

```
<html>
<head>
<title>Try Catch Example</title>
<script type="text/javascript">
try {
    eval("a ++ b");        //causes SyntaxError
```

```
} catch (oException) {
    if (oException instanceof SyntaxError) {
        alert("Syntax Error: " + oException.message);
    } else {
        alert("An unexpected error occurred: " + oException.message);
    }
}
</script>
</head>
<body>
</body>
</html>
```

Example

In this example, we do surprising things … we call a dialog box a couple of times. The purpose of this is to demonstrate to you how we can accept user input and then store that input in the JavaScript function … and continue to communicate with the end user.

```
<html>
<head>
<title>A Simple Page</title>
<script type="text/javascript">
<!--
var num1, op, num2, ans;
num1 = prompt("Enter a number:","a number");

op = prompt("Enter an operator (+, -, *, / and %)", "operator");

num2 = prompt("Enter another number:", "a number");

ans = eval(Number(num1)+ op + Number(num2));

alert(num1 + " " + op + " " + num2 + " = " + ans);

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

</body>
</html>
```

Browser Detective

This is very useful (although some folk don't like it). Because not all browsers (nor all versions) work the same, it's useful to know what is possible for your end-user's browser. Commonly Microsoft's Internet Explorer doesn't follow standards so you might want to have a cascading style sheet for one browser and one for another. In this example we get data from the browser itself (what browser is running navigator.appName)

```
<html>
```

```
<head>
<title>Browser Detective</title>
<script>
var bVersion = 0;
var isNav = false;
var isIE = false;

function checkBrowser(){
  if (navigator.appName == "Netscape"){
    isNav = true;
  } else {
    if (navigator.appName == "Microsoft Internet Explorer"){
      isIE = true;
    }
  }

  bVersion = parseInt(navigator.appVersion);

  if (bVersion < 4){
    alert("Consider getting a newer browser! This code might not work!");
  }

  if ((!isNav) && (!isIE)){
    alert("I do not recognize this browser. This code might not work");
  }
}

</script>
</head>

<body>
<center>
<h1>Browser Detective<hr></h1>
<script>
checkBrowser();

if (isNav){
  document.write("Netscape Navigator");
} else {
  if (isIE){
    document.write("Internet Explorer");
  }
}

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

Not (necessarily) for LIS488 - but you're welcome to explore …

In this example, you can switch between RGB and HSB (read the Color module). If you can read

this you should feel quite comfortable with your skill set.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
    <head>
        <title>zColor Example</title>
        <script type="text/javascript">
/*----------------------------------------------------------------------------
 * JavaScript zColor Library
 * Version 0.1
 * by Nicholas C. Zakas, http://www.nczonline.net/
 * Copyright (c) 2004-2005 Nicholas C. Zakas. All Rights Reserved.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 *----------------------------------------------------------------------------
 */


/**
 * Represents an RGB color.
 * @class
 * @scope public
 * @constructor
 * @param iRed The red value for the color (0-255)
 * @param iGreen The green value for the color (0-255)
 * @param iBlue The blue value for the color (0-255)
 */
function zRGB(iRed, iGreen, iBlue) {

    /**
     * The red value for the color.
     * @scope public
     */
    this.r /*:int */ = parseInt(iRed,10);

    /**
     * The green value for the color.
     * @scope public
     */
    this.g /*:int */ = parseInt(iGreen,10);

    /**
     * The blue value for the color.
     * @scope public
     */
```

```
    this.b /*:int */= parseInt(iBlue,10);
}

/**
 * Creates an RGB color from a hex string.
 * @scope public
 * @param sHex The hex string.
 * @return The RGB object for the hex string.
 */
zRGB.fromHexString = function (sHex /*: String */) /*:zRGB */ {

    //eliminate the pound sign
    if (sHex.charAt(0) == "#") {
        sHex = sHex.substring(1);
    } //ENd: if (sHex.charAt(0) == "#")

    //extract and convert the red, green, and blue values
    var iRed = parseInt(sHex.substring(0,2),16);
    var iGreen = parseInt(sHex.substring(2,4),16);
    var iBlue = parseInt(sHex.substring(4,6),16);

    //return an array
    return new zRGB(iRed,iGreen,iBlue);
};

/**
 * Returns a hex representation of the color.
 * @scope public
 * @return A hex representation of the color.
 */
zRGB.prototype.toHexString = function () /*:String */ {

    //extract and convert the red, green, and blue values
    var sRed = this.r.toString(16).toUpperCase();
    var sGreen = this.g.toString(16).toUpperCase();
    var sBlue = this.b.toString(16).toUpperCase();

    //make sure there are two digits in each code
    if (sRed.length == 1) {
            sRed = "0" + sRed;
    } //End: if (sRed.length == 1)
    if (sGreen.length == 1) {
            sGreen = "0" + sGreen;
    } //End: if (sGreen.length == 1)
    if (sBlue.length == 1) {
            sBlue = "0" + sBlue;
    } //End: if (sBlue.length == 1)

    //return the hex code
    return "#" + sRed + sGreen + sBlue;
};

/**
 * Returns an HSL representation of the color.
 * @scope public
 * @return An HSL representation of the color.
 */
zRGB.prototype.toHSL = function () /*:zHSL */ {
```

```
    var iMax = Math.max(this.r, this.g, this.b);
    var iMin = Math.min(this.r, this.g, this.b);
    var iDelta = iMax-iMin;

    var iLum = Math.round((iMax+iMin)/2);
    var iHue = 0;
    var iSat = 0;

    if (iDelta > 0) {
        iSat = Math.ceil(((iLum < (0.5*255)) ? iDelta/(iMax + iMin) : iDelta/((2*255)-
iMax-iMin))*255);

        var iRedC = (iMax-this.r)/iDelta;
        var iGreenC = (iMax-this.g)/iDelta;
        var iBlueC = (iMax-this.b)/iDelta;

        if (this.r == iMax) {
            iHue = iBlueC - iGreenC;
        } else if (this.g == iMax) {
            iHue = 2.0 + iRedC - iBlueC;
        } else {
            iHue = 4.0 + iGreenC - iRedC;
        }

        iHue /= 6.0;

        if (iHue < 0) {
            iHue += 1.0;
        }

        iHue = Math.round(iHue * 255);
    }

    return new zHSL(iHue,iSat,iLum);
};

/**
 * Returns the color in a string form.
 * @scope public
 * @return The color in a string form.
 */
zRGB.prototype.toString = function () /*:String */ {
    return "rgb(" + this.r + "," + this.g + "," + this.b + ")";
};

/**
 * Represents an HSL color.
 * @class
 * @scope public
 * @constructor
 * @param iHue The hue value for the color (0-255)
 * @param iSat The saturation value for the color (0-255)
 * @param iLum The luminosity value for the color (0-255)
 */
function zHSL(iHue, iSat, iLum) {

    /**
```

```
     * The hue value for the color.
     * @scope public
     */
    this.h /*:int */ = parseInt(iHue,10);

    /**
     * The saturation value for the color.
     * @scope public
     */
    this.s /*:int */ = parseInt(iSat,10);

    /**
     * The luminosity value for the color.
     * @scope public
     */
    this.l /*:int */= parseInt(iLum,10);
}

/**
 * Converts the color into RGB form.
 * @scope public
 * @return An RGB version of the color.
 */
zHSL.prototype.toRGB = function () /*:zRGB */ {

    iHue = this.h/255;
    iSat = this.s/255;
    iLum = this.l/255;

    var iRed, iGreen, iBlue;

    if (iSat == 0) {
        iRed = iGreen = iBlue = iLum;
    } else {

        var m1, m2;

        if (iLum <= 0.5) {
            m2 = iLum * (1+iSat);
        } else {
            m2 = iLum + iSat - (iLum * iSat);
        }

        m1 = 2.0 * iLum - m2;

        hf2 = iHue + 1/3;
        if (hf2 < 0) hf2 = hf2 + 1;
        if (hf2 > 1) hf2 = hf2 - 1;
        if (6 * hf2 < 1) {
            iRed = (m1+(m2-m1)*hf2*6);
        } else {
            if (2 * hf2 < 1) {
                iRed = m2;
            } else {
                if (3.0*hf2 < 2.0) {
                    iRed = (m1+(m2-m1)*((2.0/3.0)-hf2)*6.0);
                } else {
                    iRed = m1;
```

```
                }
            }
        }

        //Calculate Green
        if (iHue < 0) {iHue = iHue + 1.0;}
        if (iHue > 1) {iHue = iHue - 1.0;}
        if (6.0 * iHue < 1){
            iGreen = (m1+(m2-m1)*iHue*6.0);}
        else {
            if (2.0 * iHue < 1){
                iGreen = m2;
            } else {
                if (3.0*iHue < 2.0) {
                    iGreen = (m1+(m2-m1)*((2.0/3.0)-iHue)*6.0);
                } else {
                    iGreen = m1;
                }
            }
        }

        //Calculate Blue
        hf2=iHue-1.0/3.0;
        if (hf2 < 0) {hf2 = hf2 + 1.0;}
        if (hf2 > 1) {hf2 = hf2 - 1.0;}
        if (6.0 * hf2 < 1) {
             iBlue = (m1+(m2-m1)*hf2*6.0);
        } else {
             if (2.0 * hf2 < 1){
                 iBlue = m2;
             } else {
                 if (3.0*hf2 < 2.0) {
                     iBlue = (m1+(m2-m1)*((2.0/3.0)-hf2)*6.0);
                 } else {
                     iBlue = m1;
                 }
             }
        }

    }
    return new zRGB(Math.round(iRed*255),Math.round(iGreen*255),Math.round(iBlue*255));
};

/**
 * Returns the color in a string form.
 * @scope public
 * @return The color in a string form.
 */
zHSL.prototype.toString = function () /*:String */ {
    return "hsl(" + this.h + "," + this.s + "," + this.l + ")";
};

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

            function convertToHSL() {
                var iRed = parseInt(document.getElementById("txtRed").value);
                var iBlue = parseInt(document.getElementById("txtBlue").value);
```

```
                var iGreen = parseInt(document.getElementById("txtGreen").value);

                var oRGB = new zRGB(iRed, iGreen, iBlue);
                var oHSL = oRGB.toHSL();

                document.getElementById("txtHue").value = oHSL.h;
                document.getElementById("txtSat").value = oHSL.s;
                document.getElementById("txtLum").value = oHSL.l;

            }

            function convertToRGB() {
                var iHue = parseInt(document.getElementById("txtHue").value);
                var iSat = parseInt(document.getElementById("txtSat").value);
                var iLum = parseInt(document.getElementById("txtLum").value);

                var oHSL = new zHSL(iHue, iSat, iLum);
                var oRGB = oHSL.toRGB();

                document.getElementById("txtRed").value = oRGB.r;
                document.getElementById("txtGreen").value = oRGB.g;
                document.getElementById("txtBlue").value = oRGB.b;

            }

        </script>
    </head>
    <body>
        <form>
            <table border="0">
            <tr>
                <td>
                    R: <input type="text" id="txtRed" /><br />
                    G: <input type="text" id="txtGreen" /><br />
                    B: <input type="text" id="txtBlue" /><br />
                    <input type="button" value="to HSL -&gt;" onclick="convertToHSL()"/>
                </td>
                <td>
                    H: <input type="text" id="txtHue" /><br />
                    S: <input type="text" id="txtSat" /><br />
                    L: <input type="text" id="txtLum" /><br />
                    <input type="button" value="&lt;- to RGB" onclick="convertToRGB()"/>
                </td>
            </tr>
            </table>
        </form>

    </body>
</html>
```

Cookies

A cookie is a small bit of information stored in a text file on the user's computer by a browser.

- The cookie object is part of the Document object.
- Cookies can be created, set, and modified by setting the appropriate values of the cookie property.
- A cookie has four name attributes: expires, path, domain, and secure.
- By default, a cookie lasts only during the current browsing session.
- For a cookie to last beyond the current browsing session, the expires attribute must be set.
- The value of expires attribute can be set to any valid date string.
- The path attribute specifies the domain associated with the cookie.
- The level of association begins at the specified path and goes down into any subfolders.
- So for example, suppose http://www.java2s.com/examples/cookie.html was setting a cookie and wanted the cookie to be shared across Web pages on the java2s.com domain.
- To do this, the cookie path attribute needs to be set to "/".
- This allows the cookie to be accessed from any page on the www.java2s.com Web server.
- If the path was set to "/examples", the cookie would only be valid to pages in the examples folder and its subfolders.
- If the secure attribute is specified, the cookie will be only be transmitted over a secure channel (HTTPS).
- If secure is not specified, the cookie can be transmitted over any communications channel.

Reading a cookie

```
<html>
<head>
<title></title>
<script type=”text/javascript”>
<!--
function fillIn()
{
    if (document.cookie != "")
    {
        cookieCrumb = document.cookie.split("=")[1];
        document.form1.read1.value = cookieCrumb;
    }
    else
    {
        document.form1.read1.value = "Cookie empty!";
    }
}
//  -->
</script>
</head>
<body onload="fillIn()">
<form name="form1">
<P>The username you entered was: <input type="text" name="read1"></p>
</form>
</body>
</html>
```

The above created a cookie. Please note (an this is important) the split. Notice the = sign.

Data sent to/from a server are usually sent as "name-value pairs". The name-value pair is equivalent to declaring a variable and assigning it a value.

In this example, we write (set) a cookie and then read it back.

```
<html>
</head>
<body>
<script language="javascript">
<!--
function set_it() {
   var the_text="name=yourname&";
   var toexpire= new date("march 15, 2008");
   var expdate="expires="+toexpire.togmtstring();
   the_text+=expdate;

   var newtext=escape(the_text);
   document.cookie=newtext;
}

function read_it() {

 if (document.cookie) {

    var mycookie=document.cookie;
    var fixed_cookie= unescape(mycookie);
    var thepairs= fixed_cookie.split("&");
    var pair1= thepairs[0];
    var pair2= thepairs[1];

    var namevalue1= pair1.split("=");
    alert("read");
    alert("welcome, "+namevalue1+"!");
 } else  {
   alert("set");
   set_it();
 }
}

read_it();
//-->
</script>
</body>
</html>
```

---

this

The keyword "this" is important in object-oriented programming. When an object refers to itself, the word "this" is used. At times it isn't intuitive for us humans! In this example, a form is used to call a JavaScript … and the form needs to send data back to itself - hence the use of "this" keyword.

```
<html>
    <head>
      <title> Using this in passing form information</title>
    <script language="JavaScript">
    <!--
    function displayInfo(form){
      var myWin = open("", "","width=450,height=200");
      myWin.document.write("The defaultValue of the text box is: ");

      myWin.document.write(form.myText.defaultValue);

      myWin.document.write("<br>The name of the text area is: ");

      myWin.document.write(form.myTextArea.name);

      myWin.document.write("<br>The value of the button is: ");

      myWin.document.write(form.myButton.value);

      myWin.document.close();
    }
    -->
    </script>
    </head>
    <body>
    <form name="myForm">
      <textarea name="myTextArea" rows=2 cols=50>
      Here is some text in my text area.
      </textarea>
      <br>
      <input type=TEXT value="Change Me?" name="myText">
      <br>
      <input type=BUTTON value="Display Information" name="myButton" onClick='di
splayInfo(this.form)'>
    </form>
    </body>
</html>
```

Number Data Types

Numbers are treated differently in computers than are Strings. When a number is used in a computer program the programming language determines how much memory to save for the number. For example, the number 5 is an integer and takes one byte of memory; the number 5.1234 is a "floating point" number. The computer stores the 5 in one place and the data after the decimal (abscissa) is stored elsewhere.

- Every number in JavaScript is treated as a floating-point number.
- JavaScript supports integers, octal, hexadecimal, and so on.
- At the lowest level, JavaScript sees numbers as floating-point numbers.

```
var variable = new Number(value)
```

The Number object represents numeric value types.

You can create a Number object by specifying a value in the parameter for the number constructor.

Properties and Methods of the Number Object

Property/Method	Description
MAX_VALUE	Specifies the largest value a number can have.
MIN_VALUE	Specifies the smallest value a number can have without being equal to 0.
NaN	Stands for Not a Number. Represents a value that is not equal to any numeric value.
NEGATIVE_INFINITY	A special value that represents a negative infinity value.
POSITIVE_INFINITY	A special value that represents a positive infinity value.
prototype	Represents the prototype for the number class.
toSource()	Returns a string representation of the number object.
toString()	Returns a string representing the specified number object.
valueOf()	Returns the primitive value of a number object as a number data type.

The following example create a number object by calling the Number Constructor.

```
<html>
    <body>
      <script type="text/javascript">
    <!--
    // Creates a new number object
    var aNum = new Number(3);

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

Above all we have done is create something in memory (aNum) that is a copy (or an *instantiation*) of an Object, called "Number". Very often, objects are created by the Operating System, by the programming language, by our own programming. We need at times to test the object and to insert into or extract data from object. [See above example about the exceptions being thrown.]

It is useful to know the toString() method. Many objects have built into them a function, called toString(), that converts the value of the object (as stored in the computer's memory) into

some text we humans can read (a "String").

Parameters - primitive and reference values

This is important… we store data in a variable. Often the value of the variable is changed as the result of a function or method. For example, we might have a function called "payroll()". The payroll() function takes two parameters - the number of hours worked and the rate of pay, e.g., payroll(40, 10.72). Let's say this payroll function returns the value of the paycheck (40 * 10.72) and we store the results in a variable. E.g., var myPay = payroll(40, 10.25);

Let's say we store our actual work hours in a variable called "myHours" and the rate of pay in a variable called "myRateOfPay". Now we can switch the numbers with our variables, e.g., var myPay = myHours * myRateOfPay;

When we provide data to a function, we provide either the *original* variable and value *or* a copy of the original. This is absolutely important! [And for the computer, the reason is the data are stored in different places - on the "heap" or on the "stack" (these are separate areas in the computer's RAM).]

A variable can hold one of two types of values: primitive values and reference values.

Primitive values are data that are stored on the stack.

Primitive value is stored directly in the location that the variable accesses.

Reference values are objects that are stored in the heap.

Reference value stored in the variable location is a pointer to a location in memory where the object is stored.

Primitive types include Undefined, Null, Boolean, Number, or String.

Values in variables - nulls, etc.

You can declare a variable but not assign it a value. If there's no value then the default is "null". We must be aware of accidentally using a null value. Let's say we declare "var myHours" but don't say "= 40". For example, if we say var myPay = myHours * myRateOfPay; we would have a problem because myHours is null. The computer will try to access the variable in RAM and find there's no value there - yikes! The computer will try to calculate an empty value * myRate OfPay - the computer will freak! And generate an error. Some programming languages will generate a "NaN" error ["Not a number"]; others will send an error message ("null value"); some will just crash.

Try using these examples in a web page:

```
<code>alert(isNaN("blue"));</code> or <code>alert(isNaN("123"));</code>
```

Strings

A String is a contiguous block of characters stored in a computer's RAM. For example, the word "cat" consists of three letters c + a + t. Because we are looking at more than one character, we have a data type called String. If we declare a variable called "animal" and want to store the word "cat", we are actually creating an instantiation of a String object and inserting into that object the word "cat": var animal = "cat";

The computer keeps track of address in memory the first letter only - this is called a pointer. Let's there's a grid with a starting address of (0, 0) [the horizontal value is 0 and the vertical value is 0].

So at the starting point of (0,0) we store the letters "c a t". So [0,0] has "c", [0,1] has "a", [0,2] has "t". The computer keeps track of (0,0) for the location of the String animal. In some computer languages, such as C++, we store the data via a "pointer" and get the data back via "dereferencing". [In other words, we point to the address in memory, hence we have a variable called a pointer; and we get back the data by "dereferencing" or asking the pointer to fetch the data at that address.]

Strings: String objects have lots of built-in methods. For example, we might have a String of letters and we want to convert them all to upper case. E.g.,

```
var myString = "cat";       //   myString = "cat";
```

to convert this string to all capitals, we can inherit the "toUpperCase()" method from the String object.

```
var myOtherString = myString.toUpperCase();
```

myOtherString now equals "CAT".

But we can make a String refer to itself! var myString = myString.toUpperCase();

There are a number of String related functions built into every language. One of the most useful is to compare two strings. [For instance, this is useful when checking a user's login and password!]

```
<html>
<body>
<script language="javascript">
<!--
var myvar="aa";


if(myvar=="aa") {
    window.alert("aa");
} else  {
    window.alert("not aa");
}

//-->
```

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

==========

Functions

- All function declaration must begin with the keyword function followed by the name of the function.
- Parentheses are placed after the function name to hold arguments.
- If more than one argument are used, use commas to separate the arguments.
- If no arguments, leave the space between the parentheses empty.
- Curly brackets are used to contain the code related to the function.
- Curly brackets are not optional.
- JavaScript uses call by value.

The heart-and-soul of programs is the function. Functions do things - they are the action of a program! You can always identify a function in your readings by the ( ). Below is an example of a function (called "test()") that accepts two parameters, a String and a Number).

```
<html>
<script language='javascript'>
<!--
    var astring = "a"
    var anumber = 1;
    function test(astring, anumber) {
      astring = "b";
      anumber = 2;
      document.write("<br><br><br><br><br>during function call:<br>");
      document.write("astringcopy=",astring,"<br>");
      document.write("anumbercopy=",anumber,"<br>");
    }

    document.write("<br><br><br><br><br>before function call:<br>");
    document.write("astring=",astring,"<br>");
    document.write("anumber=",anumber,"<br>");


    test(astring,anumber);


    document.write("<br><br><br><br><br><br><br>after function call:<br>");
    document.write("astring=",astring,"<br>");
    document.write("anumber=",anumber,"<br>");
    //-->
</script>
</html>
```

Note this example. here we are defining *our own* function called "myMessage()". Notice how in

our function we are using also a function that's built into JavaScript (the "alert()"). Now, in our html page we must refer to "myMessage()"

```
<html>
<head>
<title>A Simple Page</title>
<script language="JavaScript">
<!--
function yourMessage()
{
    alert("Your first function!");
}
//  -->
</script>
</head>
<body>

</body>
</html>
```

Having borrowed JavaScript's alert() function, let's integrate some things - a variable, too…

```
<html>
<head>
<title>Function Demo</title>

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

function greetVisitor()
{
     var myName = prompt("Name.", "");
     alert("Welcome " + myName + "!")
}

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

<h1>Function Demo</h1>

</body></html>
```

Math functions

[tba]

Forms

An HTML form is defined by using the

element, which has several attributes:

1. method -- Indicates whether the browser should sent a GET request or a POST request
2. action -- Indicates the URL to which the form should be submitted
3. enctype -- The way the data should be encoded when sent to the server. The default is application/x-www-url-encoded, but it may be set to multipart/form-data if the form is uploading a file.
4. accept -- Lists the mime types the server will handle correctly when a file is uploaded
5. accept-charset -- Lists the character encodings that are accepted by the server when data is submitted

http://www.java2s.com/Tutorial/JavaScript/0200__Form/GettingformReferences.htm