Internet Programming

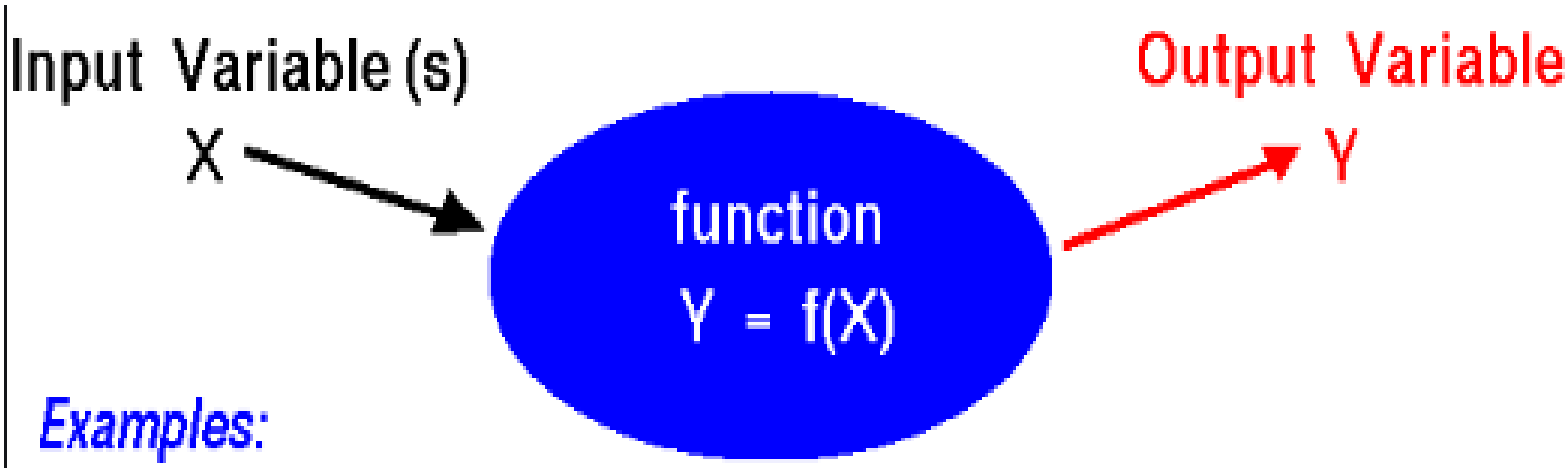
JavaScript Functions, Objects, Arrays

Dan Brandon, Ph.D., PMP

Functions

- There are many built in functions, and these are associated with built in (standard) objects:
  - object.function (…);
- Some objects and functions are relatively “standard”, and some are browser specific
- We have already used several object’s functions: document, window, Date, …
- You can define your own functions and objects also !

Functions (con't)



Examples:

$\sin(x)$ $\exp(x)$ e^x $x^3 + x^2 + 5x + 12$

$\cos(x)$ $\ln(x)$ $\sqrt[2]{x}$ $\cosh(x)$

$\tan(x)$ $\tan^{-1}(x)$ $x!$

User Functions

- You can make user defined functions, just like in C/C++ or Java:
  - function myFunction(optional arguments)
  - {
    - ...code...
    - return (optionally a variable)
  - }
- Function scripts can be placed anywhere, but are best placed in the head section
- Functions are the normal (and recommended) way to use JS

User Functions (con't)

- Example:
  - function multFunc (n1, n2)
    {
    - n3 = n1 * n2;
    - return n3;
  - }

Functions (con't)

```
function multFunc (n1, n2) {
  n3 = n1 * n2;
  return n3;
}
```

- Unlike C/C++/Java, there is no return type specified nor argument typing
- Variables defined within a function are local to that function and cannot be seen outside (i.e. n3 in above example)
- Functions are called (invoked) as:
  - z = multFunc(x, y);
- Function arguments of type numeric or Boolean are passed by value (function makes a separate copy); string, array, and object arguments are passed by reference (address)

Class Exercise

- Rewrite the previous class exercise of two input numbers being multiplied together and then result displayed
  - Can use your version with or without error checking
- In this new version use a function [i.e. “mult (n1, n2)”] to do the multiplication

- Try to work class exercises without looking ahead !

Exercise:

```
<HTML>
<HEAD><TITLE>Function</TITLE>
</HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
     function mult (x1, x2) {
            return x1 * x2;
     }
     var n1, n2, s1, s2;
     s1 = window.prompt("Enter first number:", "0");
     s2 = window.prompt("Enter second number:", "0");
     n1 = parseInt(s1);
     n2 = parseInt(s2);
     var n3 = mult (n1, n2);
     document.writeln("<H1>The product of these numbers is:
" + n3 + "</H1>");
// -->
</SCRIPT>
<BODY></BODY></HTML>
```

```
<HTML>
<HEAD><TITLE>Function</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
     function mult (x1, x2) {
            return x1 * x2;
     }
// -->
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
<!--
     var n1, n2, s1, s2;
     s1 = window.prompt("Enter first number:", "0");
     s2 = window.prompt("Enter second number:", "0");
     n1 = parseInt(s1);
     n2 = parseInt(s2);
     var n3 = mult (n1, n2);
     document.writeln("<H1>The product of these numbers is:
" + n3 + "</H1>");
// -->
</SCRIPT>
</BODY></HTML>
```

A Better Way, place JS function in HEAD

Global Scope

- Variables declared Globally (outside any function) have Global Scope.
  - Example
  - var carName = "Volvo";
  - // code here can use carName
  - …
  - function myFunction() {
  - // code here can also use carName
  - }

Function Scope

- Variables declared Locally (inside a function) have Function Scope.
- Example
  - // code here can NOT use carName
  - …
  - function myFunction() {
  - var carName = "Volvo";
  - // code here CAN use carName
  - }
  - // code here can NOT use carName

Block Scope

- Variables declared with the var keyword cannot have Block Scope
- Variables declared inside a block {} can be accessed from outside the block, example:
- {
- var x = 2;
- }
- // x CAN be used here
- Variables declared with the let keyword can have Block Scope
- Variables declared inside a block {} cannot be accessed from outside the block, example:
- {
- let x = 2;
- }
- // x can NOT be used here

JS Global Functions
[a part of JS's global object]

- var s1 = escape(s2)
  - Encodes string s2 into string s1 replacing all punctuation and non-ASCII characters with hexadecimal codes
- eval(s1)
  - Takes the string representation of JS code (s1) and executes it; provides for dynamic JS code generation
- isFinite(n1)
  - Returns true if n1 is within allowable number range

JS Global Functions (con't)

- isNaN(n1)
  - Returns true if n1 is not a number
- var n1 = parseFloat(s1)
  - Converts string to float, returns NaN if s1 is not numeric
- var n1 = parseInt(s1)
  - Converts string to integer, returns NaN if s1 is not numeric
- var s1 = unescape (s2)
  - Opposite of escape

Timing Functions

- setInterval (function, interval) – this function will invoke another function over and over again every so many milliseconds
- setTimeout(function, interval) – same as setInterval, but will only invoke the function one time
- Example:
  - window.setInterval(“myfun()”, 100);

Timing Functions (con't)

- **Time-delayed command** - run after a specified amount of time has passed
- Time delay is defined using the following:

  `setTimeout("command", delay);`

  where *command* is a JavaScript command and *delay* is the delay time in milliseconds before a browser runs the command
- The command must be placed within either double or single quotation marks

Timing Functions (con't)

- To cancel a time-delayed command:

```
clearTimeout();
```

- Browser can process unlimited number of time-delayed commands
- Can assign a unique identification number to distinguish commands

```
var timeID = setTimeout("command", delay);
```

where *timeID* is a variable that stores the ID of the time-delayed command

Timing Functions (con't)

- A timed-interval command instructs browsers to run a command repeatedly at a specified interval
- Timed-interval command syntax:

```
setInterval("command", interval);
```

  where *interval* is the interval in milliseconds before the command is run again

- Timed-interval commands are halted using the following statement:

```
clearInterval();
```

Timing Functions (con't)

- Distinguish one timed-interval command from another by storing the time ID in a variable

```
var timeID = setInterval("command",
interval);
```

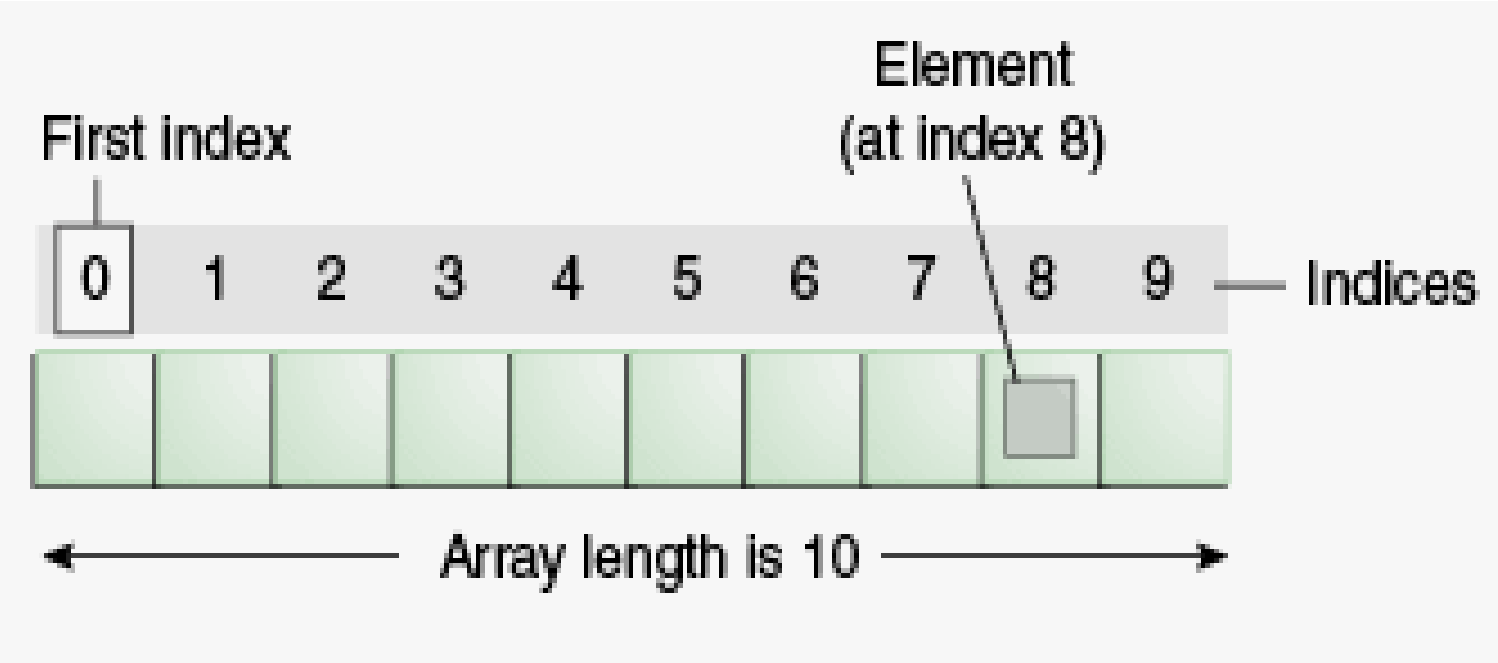
- Halt the timed-interval command by applying the `clearInterval()` method with *`timeID`* as the parameter value:

```
clearInterval(timeID);
```

Arrays

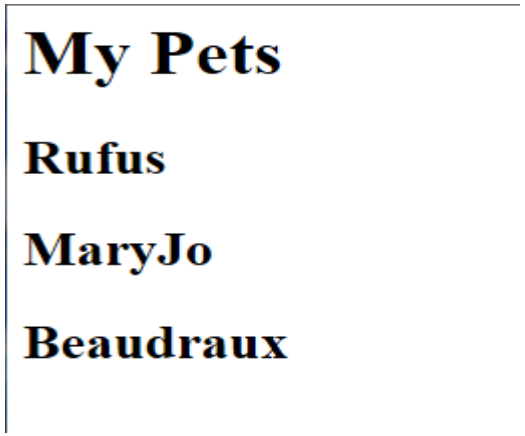
- You can have arrays of numbers, strings, or even objects
  - var a = new Array (100); // a is starting address
  - var b = new Array (); // no initial size specified
  - var c = new Array (“cyan”, “magenta”, “yellow”);
  - var d = [3, 6, 9, 12];
  - var e = [3, , , 12]; // not all elements known initially
- Array index numbering starts at zero:
  - x[0] = …;
- Arrays can grow (no need to use ‘malloc’ or ‘new’ like in C/C++)

Arrays (con't)


Element
(at index 8)
First index
0
1
2
3
4
5
6
7
8
9
Indices
Array length is 10

Array & Loop Example

```
<HTML>
<HEAD><TITLE>Array Example</TITLE>
</HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
     document.writeln("<H1>My Pets</H1>");
     var names = new Array ("Rufus", "MaryJo", "Beaudraux");
     for (i=0; i<names.length; i++) {
          document.writeln("<P><H2>" + names[i] +
"</H2>");
     }
// -->
</SCRIPT>
<BODY></BODY></HTML>
```



Class Exercise

- Extend the previous example to include an age for each pet (in another array), and display the information using an HTML table:

- **My Pets**
  - **Name Age**
  - **Rufus 3**
  - **MaryJo 5**
  - **Beaudraux 2**

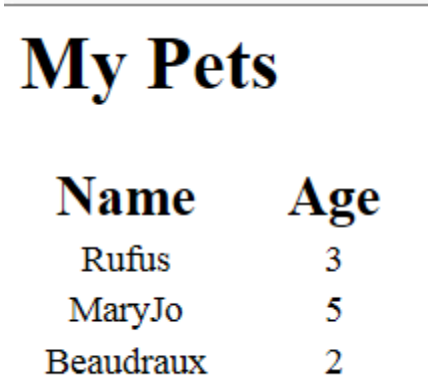
- **Hint follows…**

Hint:

- `var names = new Array ("Rufus", "MaryJo", "Beaudraux");`

- `var ages = [3, 5, 2];`

- Try to work class exercises without looking ahead !




```
<HTML>
<HEAD><TITLE>Array Example</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
      var names = new Array ("Rufus", "MaryJo", "Beaudraux");
      var ages = [3, 5, 2];
      document.writeln("<H1>My Pets</H1>");
      document.writeln("<TABLE><TR Align='Center'><TH
Width='100'><H2>Name</H2></TH>”);
       document.writeln(“<TH Width = '100'><H2>Age</H2></TH></TR><H3>");
      for (i=0; i<names.length; i++) {
              document.writeln("<TR Align='Center'><TD>" + names[i]);
                document.writeln("</TD><TD>" + ages[i] + "</TD></TR>");
      }
      document.writeln("</H3></TABLE>");
// -->
</SCRIPT></HEAD>
<BODY></BODY></HTML>
```

ArrayExample2.html

Mixing HTML & JS

```
<HTML>
<HEAD><TITLE>Array Example</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
     var names = new Array ("Rufus", "MaryJo", "Beaudraux");
     var ages = [3, 5, 2];
// -->
</SCRIPT>
</HEAD>
<BODY>
<H1>My Pets</H1>
<TABLE><TR Align='Center'><TH Width='100'><H2>Name</H2></TH><TH
Width = '100'><H2>Age</H2></TH></TR><H3>
<SCRIPT LANGUAGE="JavaScript">
<!--
     for (i=0; i<names.length; i++) {
               document.writeln("<TR Align='Center'><TD>" + names[i]
+ "</TD><TD>" + ages[i] + "</TD></TR>");
     }
// -->
</SCRIPT>
</H3></TABLE>
</BODY></HTML>
```

ArrayExample3.html

Using JS Functions & Arrays

```
<HTML>
<HEAD><TITLE>Array Example</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
      var names = new Array ("Rufus", "MaryJo", "Beaudraux");
      var ages = [3, 5, 2];
      function writeRow(n, a) {
                document.writeln("<TR Align='Center'><TD>" + n + "</TD><TD>" + a
                + "</TD></TR>");
      }
// -->
</SCRIPT>
</HEAD>
<BODY>
<H1>My Pets</H1>
<TABLE><TR Align='Center'><TH Width='100'><H2>Name</H2></TH><TH Width =
'100'><H2>Age</H2></TH></TR><H3>
<SCRIPT LANGUAGE="JavaScript">
<!--
      for (i=0; i<names.length; i++) {
                writeRow (names[i], ages[i]);
      }
// -->
</SCRIPT>
</H3></TABLE>
</BODY></HTML>
```

ArrayExample4.html

Image Arrays

- One can have the images in a page stored in an array
- This also speeds up the image loading process, by “preloading” the images:
  - <SCRIPT LANGUAGE=“JavaScript”>
    - If (document.images) {
      - images[0]=new Image(); images[0].src=“myImg0.gif”;
      - images[1]=new Image(); images[1].src=“myImg1.gif”;
      - images[2]=new Image(); images[2].src=“myImg2.gif”;
    - }
  - </SCRIPT>

Mixing Data Types

- Array values need not be of the same data type
- Mix of numeric values, text strings, and other data types within a single array is allowed, but not recommended
- Example

```
var x = [“April”, 3.14, true, null];
```

Changing Order

- Items are placed in an array either in the order in which they are defined or explicitly by index number, by default
- JavaScript supports two methods for changing the order of the array items
  - `reverse()`
  - `sort()`
- `reverse()`: Reverses the order of items in an array, making the last items first and the first items last

Changing Order (con't)

- The sort() method casts elements to strings and compares the strings to determine the orders
- Consider the following example:
  - numbers = [0, 1 , 2, 3, 10, 20, 30];
  - numbers.sort();
  - console.log(numbers);
- The output is:
  - [0, 1, 10, 2, 20, 3, 30]
- In this example, the sort() method places 10 before 2 because the string "10" comes before "2" when doing a string comparison

Numerical Sort

- **Compare function**: Compares values of two adjacent array items
- General form of a compare function is

```
function fname(a, b) {
return a negative, positive, or 0 value
}
```

where `fname` is the name of the compare function and `a` and `b` are parameters that represent a pair of array values

Numerical Sort (con't)

- Based on comparison of two adjacent array item values, the function returns a negative, positive, or zero value
  - If a negative value is returned, then `a` is placed before `b` in the array
  - If a positive value is returned, then `b` is placed before `a`
  - If a zero value is returned, `a` and `b` retain their original positions

Numerical Sort (con't)

- Function to sort numeric values in ascending order

```
function ascending(a, b) {
return a - b;
}
```

- Function to sort numbers in descending order

```
function descending(a, b) {
return b - a;
}
```

Numerical Sort (con't)

- The compare function is applied to the `sort()` method as follows

  ```
  array.sort(fname)
  ```

  where `fname` is the name of the compare function
- Example

  ```
  x.sort(ascending)
  ```

Numerical Sort (con't)

- Using the same example as previously, to sort numerically:
  - numbers = [0, 1 , 2, 3, 10, 20, 30];
  - numbers.sort(function(a , b){
  - if(a > b) return 1;
  - if(a < b) return -1;
  - return 0;
  - });
  - console.log(numbers);
- Output is:
  - [0, 1, 2, 3, 10, 20, 30]

Numerical Sort (con't)

- using the arrow function syntax:

- let numbers = [0, 1 , 2, 3, 10, 20, 30];
- numbers.sort((a,b) => {
- if(a > b) return 1;
- if(a < b) return -1;
- return 0;
- });

Extracting and Inserting

- **Subarray**: Section of an array
- To create a subarray use `slice()` method

  `array.slice(start, stop)`

  where `start` is the index value of the array item at which the slicing starts and `stop` is the index value at which the slicing ends
- The `stop` value is optional; if it is omitted, the array is sliced to its end

Extracting and Inserting (con't)

- `splice()`: Removes and inserts array items

  `array.splice(start, size, values)`

  `start` is the starting index in the array, `size` is the number of array items to remove after the `start` index, and `values` is an optional comma-separated list of values to insert into the array
- If no `values` are specified, the splice method simply removes items from the array
- Always alters the original array

Data Structures

- **Stack**: Arrays can be used to store information in a data structure
- New items are added to the top of the stack or to the end of the array
- A stack data structure employs the **last-in first-out (LIFO)** principle
- In the **LIFO** principle the last items added to the stack are the first ones removed

Data Structures (con't)

- `push()` method: Appends new items to the end of an array

  `array.push(values)`

  where `values` is a comma-separated list of values to be appended to the end of the array
- `pop()` method: Removes or **unstacks** the last item

  `array.pop()`

Data Structures (con't)

- **Queue:** Employs the **first-in-first-out (FIFO)** principle in which the first item added to the data list is the first removed
- Similar to a stack
- `shift()` method: Removes the first array item
- `unshift()` method: Inserts new items at the front of the array

Data Structures (con't)

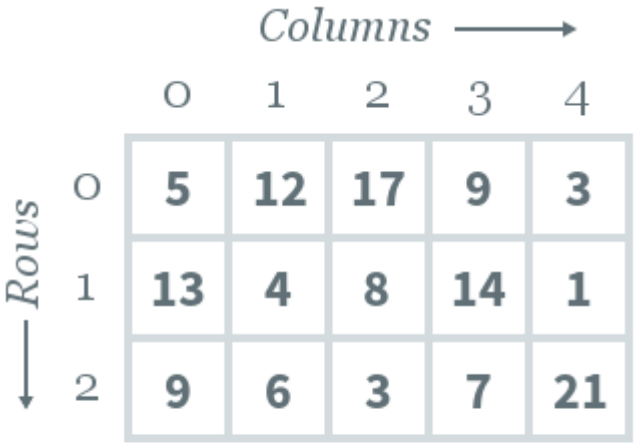
Method	Description
`copyWithin(target, start[, end])`	Copies items within the array to the *target* index, starting with the *start* index and ending with the optional *end* index
`concat(array1, array2,...)`	Joins the array to two or more arrays, creating a single array containing the items from all the arrays
`fill(value[, start][, end])`	Fills the array with items having the value , starting from the *start* index and ending at the *end* index
`indexOf(value[, start])`	Searches the array, returning the index number of the first element equal to *value*, starting from the optional *start* index
`join(separator)`	Joins all items in the array into a single text string; the array items are separated using the text in the *separator* parameter; if no *separator* is specified, a comma is used
`lastIndexOf(value[, start])`	Searches backward through the array, returning the index number of the first element equal to *value*, starting from the optional *start* index
`pop()`	Removes the last item from the array
`push(values)`	Appends the array with new items, where *values* is a comma-separated list of item values

Data Structures (con't)

Method	Description
`reverse()`	Reverses the order of items in the array
`shift()`	Removes the first item from the array
`slice(start, stop)`	Extracts the array items starting with the *start* index up to the *stop* index, returning a new subarray
`array.splice(start, size, values)`	Extracts *size* items from the array starting with the item with the index *start*; to insert new items into the array, specify the array items in a comma-separated *values* list
`array.sort(fname)`	Sorts the array where *fname* is the name of a function that returns a positive, negative, or 0 value; if no function is specified, *array* is sorted in alphabetical order
`array.toString()`	Converts the contents of the array to a text string with the array values in a comma-separated list
`array.unshift(values)`	Inserts new items at the start of the array, where *values* is a comma-separated list of new values

Multi-Dimensional Arrays

- Define and initialize:
  - var items = [[1, 2],[3, 4],[5, 6]];
  - items[i][j] = k;
- Define (array of arrays):
  - var x = new Array(10);
  - for (var i = 0; i < 10; i++) {
- x[i] = new Array(20);
  - }
- Set values:
  - x[5][12] = 3.0;



2D Array of size 3 x 5

Callbacks for Array Processing

- JavaScript supports several methods to loop through the contents of an array without having to create a program loop structure
- The methods are faster than program loops but may not be supported in older browsers
- Each of these methods is based on calling a function that will be applied to each item in the array

Callbacks (con't)

- The general syntax

  `array.method(callback [, thisArg])`

  where `array` is the array, `method` is the array method, and `callback` is the name of the function that will be applied to each array item
- `thisArg:` A `callback` optional argument that can be included to pass a value to the function

Callbacks (con't)

- General syntax of the callback function

```
function callback(value [, index,
array]) {
commands }
```

  where `value` is the value of the array item during each pass through the array, `index` is the numeric index of the current array item, and `array` is the name of the array

- Only the `value` parameter is required; the other are optional

Callbacks (con't)

- `forEach()` method: Runs a function for each item in the array
- General syntax

  ```
  array.forEach(callback [, thisArg])
  ```

  where `callback` is the function that is applied to each item in the array

Callbacks (con't)

- `map()` method: The function it calls returns a value that can be used to map the contents of an existing array into a new array
- Example

```
var x = [3, 8, 12];
var y = x.map(DoubleIt);
function DoubleIt(value) {
return 2*value;
}
```

Callbacks (con't)

- `filter()` method: Extracts array items that match some specified condition

  `array.filter(callback [, thisArg])`

  where `callback` is a function that returns a Boolean value of `true` or `false` for each item in the array
- Array items that return a value of `true` are copied into the new array

Callbacks (con't)

- `every()` method: Returns the value `true` if every item in the array matches the condition specified by the callback function; if otherwise, returns `false`

  ```
  array.every(callback [, thisArg])
  ```

- `some()` method: Returns a value of `true` if some array items match a condition specified in the function, but otherwise returns `false` if none of the array items match the condition specified in the function

Callbacks (con't)

Array Method	Description
every(*callback* [, *thisArg*])	Tests whether the condition returned by the *callback* function holds for all items in *array*; in all array methods, the optional *thisArg* parameter is used to pass values to the *callback* function
filter(*callback* [, *thisArg*])	Creates a new array populated with the elements of *array* that return a value of *true* from the *callback* function
forEach(*callback* [, *thisArg*])	Applies the *callback* function to each item in *array*
map(*callback* [, *thisArg*])	Creates a new array by passing the original array items to the *callback* function, which returns the mapped value of the array items
reduce(*callback* [, *thisArg*])	Reduces *array* by keeping only those items that return a value of *true* from the *callback* function

Callbacks (con't)

Array Method	Description
`reduceRight(callback [, thisArg])`	Reduces *array* from the last element by keeping only those items that return a value of *true* from the *callback* function
`some(callback [, thisArg])`	Tests whether the condition returned by the *callback* function holds for at least one item in *array*
`find(callback [, thisArg])`	Returns the value of the first element in the array that passes a test in the *callback* function
`findIndex(callback [, thisArg])`	Returns the index of the first element in the array that passes a test in the *callback* function

Standard Object Types

- JS has a number of standard object types (including the browser itself) all of which have methods and properties
- Later we will study the “Document Object Model” in more detail and the standard types involving the HTML document
- The appendix of a later lesson describes the standard JS objects with their properties and methods (except for Date, String, and Math covered in this lesson)

Math Object

- **Math object**: Built-in object used to perform mathematical tasks and store mathematical values
- Syntax to apply a Math method:

  `Math.`*`method`*`(`*`expression`*`)`

  where *`method`* is the method applied to a mathematical expression

Math Object (con't)

Method	Description	Example	Returns
`Math.abs(x)`	Returns the absolute value of *x*	`Math.abs(-5)`	5
`Math.ceil(x)`	Rounds *x* up to the next highest integer	`Math.ceil(3.58)`	4
`Math.exp(x)`	Raises *e* to the power of *x*	`Math.exp(2)`	e^2 (approximately 7.389)
`Math.floor(x)`	Rounds *x* down to the next lowest integer	`Math.floor(3.58)`	3
`Math.log(x)`	Returns the natural logarithm of *x*	`Math.log(2)`	0.693
`Math.max(x, y)`	Returns the larger of *x* and *y*	`Math.max(3, 5)`	5

Math Object (con't)

Method	Description	Example	Returns
`Math.min(x, y)`	Returns the smaller of *x* and *y*	`Math.min(3, 5)`	3
`Math.pow(x, y)`	Returns *x* raised to the power of *y*	`Math.pow(2,3)`	2^3 (or 8)
`Math.rand()`	Returns a random number between 0 and 1	`Math.rand()`	Random number between 0 and 1
`Math.round(x)`	Rounds *x* to the nearest integer	`Math.round(3.58)`	4
`Math.sqrt(x)`	Returns the square root of *x*	`Math.sqrt(2)`	approximately 1.414

Math Object Methods (con't)

- Usage:
  - document.writeln(Math.sqrt(x));
- For random numbers use:
  - var x = Math.floor(1 + Math.random() * z);
  - x will be a random number between 1 and z
- A number of constants are also defined in the Math object:
  - Math.E, Math.LN2, Math.LN10, Math.PI, …

Math Constants

- Syntax to access mathematical constants is

  `Math.`*`CONSTANT`*

  where *`CONSTANT`* is the name of one of the mathematical constants supported by `Math` object

Constant	Description
Math.E	The base of the natural logarithms (2.71828…)
Math.LN10	The natural logarithm of 10 (2.3026…)
Math.LN2	The natural logarithm of 2 (0.6931…)
Math.LOG10E	The base 10 logarithm of *e* (0.4343…)
Math.LOG2E	The base 2 logarithm of *e* (1.4427…)
Math.PI	The value of π (3.14159…)
Math.SQRT1_2	The value of 1 divided by the square root of 2 (0.7071…)
Math.SQRT2	The square root of 2 (1.4142 …)

Common String Object Methods

charAt(index)	return character in string
charCodeAt()	return Unicode in string
concat()	concatenation
indexOf(char [,offset])	returns index of char in string
lastIndexOf(char [,offset])	returns index from rear
split(delimiter)	parse String into tokens *
substr(start, length)	returns substring
substring(start, end)	returns substring
toLowerCase()	makes string lower case
toString()	makes another object a String
toUpperCase()	makes string upper case
tt()	makes string monospaced (teletype)

- * ‘strtok’ in C/C++, or StringTokenizer in Java

String Methods that Generate HTML [wrap object string in HTML tags]

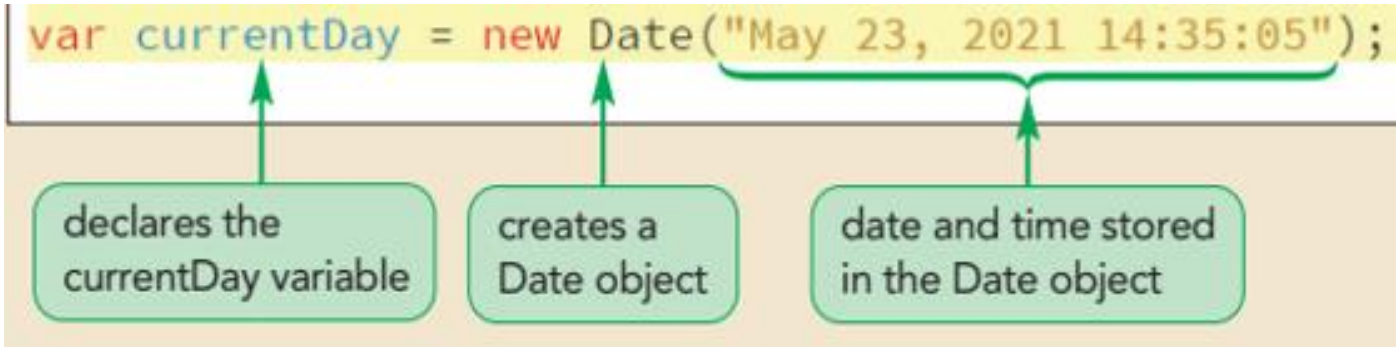
big()	make string a larger font
blink()	make string blink
bold()	makes string bold
fixed()	makes teletype string
fontcolor(color)	sets color of string
fontsize(size)	sets size of string
italics()	makes Italics tag set
link()	wraps in anchor tags
small()	use smaller font for string
strike()	wraps in STRIKE tag
sub()	make string a subscript
sup()	makes string a superscript

Date Object

- **Date object**: Built-in JavaScript object used to store information about dates and times
- Defined using the following expression:

```
new Date("month day, year hrs:mins:secs");
```

where *month*, *day*, *year*, *hrs*, *mins*, and *secs* provide the Date object's date and time



Common Date Object Methods

getDate()	returns day of month in Date
detDay()	returns day of week
getMinutes()	return minutes
getSeconds()	returns seconds
getTime()	returns milliseconds past Epoch
getTimezoneOffset()	returns diff to GMT in min.
getYear()	returns year in Date
toString()	returns day, date, time
toLocalString()	returns day, date, time in local format

Common Date Object Methods (con't)

setDate(day)	set the month day (1-31)
setHours(hours)	set the hours (0-23)
setMinutes(min)	set minutes (0-59)
setMonth(month)	set month (1-12)
setSeconds(sec)	set seconds (1-59)
setYear(year)	set year of date

Creating a Current Date

- Creating a current date object:
  - var x = new Date();
- Displaying the full date/time in text:
  - document.writeln(“<H2>Current time: “ + x.toLocaleString() + “</H2>”);

Displaying a Date

- `var thisDay = new Date("May 23, 2021 14:35:05");`

Method	Description	Results
thisDay.getSeconds()	seconds	5
thisDay.getMinutes()	minutes	35
thisDay.getHours()	hours	14
thisDay.getDate()	day of the month	23
thisDay.getMonth()	month number, where January = 0, February =1, etc.	4

Displaying a Date (con't)

- `var thisDay = new Date("May 23, 2021 14:35:05");`

Method	Description	Results
thisDay.getFullYear()	year	2021
thisDay.getDay()	day of the week, where Sunday = 0, Monday = 1, etc.	0
thisDay.toLocaleDateString()	text of the date using local conventions	"5/23/2021"
thisDay.toLocaleTimeString()	text of the time using local conventions	"2:35:05 PM"

Date Set

Date Method	Description
date.setDate(*value*)	Sets the day of the month of *date*, where *value* is an integer, ranging from 1 up to 31 (for some months)
date.setFullYear(*value*)	Sets the four-digit year value of *date*, where *value* is an integer
date.setHours(*value*)	Sets the 24-hour value of *date*, where *value* is an integer ranging from 0 to 23
date.setMilliseconds(*value*)	Sets the millisecond value of *date*, where *value* is an integer between 9 and 999
date.setMinutes(*value*)	Sets the minutes value of *date*, where *value* is an integer ranging from 0 to 59
date.setMonth(*value*)	Sets the month value of *date*, where *value* is an integer ranging from 0 (January) to 11 (December)
date.setSeconds(*value*)	Sets the seconds value of *date*, where *value* is an integer ranging from 0 to 59
date.setTime(*value*)	Sets the time value of *date*, where *value* is an integer representing the number of milliseconds since midnight on January 1, 1970

Methods of Number Object

- To create a number object:
  - var x = new Number(x);
  - This creates a “wrapper” object for the intrinsic number x (literal or symbolic)
- Methods in this class include:
  - toString() Returns a string representation of the number

    valueOf() Returns the intrinsic numeric value

  Constants include: Number.NAN (not a number), Number.MAX_VALUE, Number.MIN_Value

Methods of Boolean Object

- To create a Boolean object:
  - var x = new Boolean(y)
  - If y is false, zero, null, Number.NAN, empty string (“”), then x is false; otherwise x is true
- Methods of this class:
  - toString() returns “true” or “false”
  - valueOf() returns true if the Boolean object

    is true, otherwise false

Making Your Own Function Libraries

```
// Trim functions (returns strings with blanks trimmed)
function LeftTrim(str){
        for(var i=0;str.charAt(i)==" ";i++);
        return str.substring(i,str.length);
}
function RightTrim(str){
        for(var i=str.length-1;str.charAt(i)==" ";i--);
        return str.substring(0,i+1);
}
function Trim(str){
        return LeftTrim(RightTrim(str));
}
```

Regular Expressions in JavaScript

- Recently, these capabilities were introduced to JavaScript
- A regular expression is just a sequence or a pattern of characters that is matched against a string of text when performing searches and replacements
- Many tasks can be done with regular expressions; the most common one is to find out whether a given string matches a particular pattern
- You can use a substitution command to replace matching sections with another string of your choice

Trim Functions Using Regular Expressions

- function ltrim (s) {
  - return s.replace(/^\s*/, "");
- }
- function rtrim (s) {
  - return s.replace(/\s*$/, "");
- }
- function trim (s) { return rtrim(ltrim(s)); }

Not so “courteous” functions !

Window Function to Add a “Favorite”

```
<HTML>
<HEAD><TITLE>Adding a Bookmark in JavaScript</TITLE>
</HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
    window.external.addFavorite('http://www.cbu.edu/business','CBU School of Business');
// -->
</SCRIPT>
<BODY></BODY></HTML>
```

References

- Web Coding & Development All-in-One For Dummies (For Dummies (Computer/Tech)) by Paul McFedries
- Get Coding!: Learn HTML, CSS & JavaScript & Build a Website, App & Game by Young Rewired State
- Web Design with HTML, CSS, JavaScript and jQuery Set by Jon Duckett

Homework

- Textbook Chapter 10
- Using JS and HTML (ParseExample.html):
  - Ask the user for a sentence
  - Break the sentence down into words
  - List each word found in an HTML table
- Hybrid online reading assignment
  - Appendix on JavaScript user defined objects

Appendix

User Defined Object Types

User Defined Object Types

- Objects are created by using the “new” statement
- A function has to be written to serve as the “constructor” for the object (using “this”):
  - function Person(n, a) {
    - this.name = n;
    - this.age = a;
    - return this;
  - }
- To “instantiate” an object of type person:
  - p1 = new Person(“John”, 34);

User Defined Objects (con't)

- To create object functions (methods):
  - define the function
  - include the function in the constructor
- Example:
  - function showInfo() {
    - info = "Name: " + this.name + " Age: " + this.age;
    - alert(info);
  - }
  - function Person(name, age) {
    - this.name = name;
    - this.age = age;
    - this.showInfo = showInfo;
    - return this; }

User Defined Objects (con't)

- Object properties and methods are referenced with the dot notation directly:
  - p1 = new Person("John", 34);
  - a = p1.age
  - n = p1.name;
  - p1.showInfo();
- Or within a block of code using "with":
  - with(p1) {
    - a = age;
    - n = name;
  - }

Class Exercise

- Modify the previous class exercise to utilize a user defined object called “Pet” (each pet has a name and age).
- Create an array of pets, and then loop thru the array to create the HTML table.
- Hint follows…

Hint:

```
function Pet (n, a) {
        this.name = n;
        this.age = a;
        return this;
}
```

- Try to work class exercises without looking ahead !

```
<HTML>
<HEAD><TITLE>Object Example</TITLE>
</HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
      function Pet (n, a) {
          this.name = n;
          this.age = a;
      return this;
      }
      var p1 = new Pet("Rufus", 3);
      var p2 = new Pet("MaryJo", 5);
      var p3 = new Pet("Beaudraux", 2);
      var pets = [p1, p2, p3];
      document.writeln("<H1>My Pets</H1>");
      document.writeln("<TABLE><TR Align='Center'><TH
Width='100'><H2>Name</H2></TH><TH Width =
'100'><H2>Age</H2></TH></TD><H3>");
      for (i=0; i<pets.length; i++) {
                document.writeln("<TR Align='Center'><TD>" +
pets[i].name + "</TD><TD>" + pets[i].age + "</TD></TR>");
      }
      document.writeln("</H3></TABLE>");
// -->
</SCRIPT>
<BODY></BODY></HTML>
```

ObjectExample.html

Using an Object Function (Method)

```
<HTML>
<HEAD><TITLE>Object Example</TITLE>
</HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
        function writeRow() {
                document.writeln("<TR Align='Center'><TD>" + this.name + "</TD><TD>" + this.age +
"</TD></TR>");
        }
        function Pet (n, a) {
                this.name = n;
                this.age = a;
                this.writeRow = writeRow;
        return this;
        }
        var p1 = new Pet("Rufus", 3);
        var p2 = new Pet("MaryJo", 5);
        var p3 = new Pet("Beaudraux", 2);
        var pets = [p1, p2, p3];
        document.writeln("<H1>My Pets</H1>");
        document.writeln("<TABLE><TR Align='Center'><TH Width='100'><H2>Name</H2></TH><TH Width =
'100'><H2>Age</H2></TH></TD><H3>");
        for (i=0; i<pets.length; i++) {
                pets[i].writeRow();
        }
        document.writeln("</H3></TABLE>");
// -->
</SCRIPT>
<BODY></BODY></HTML>
```

Using Objects for Arrays

- You can also treat arrays like user defined objects and make constructors (in JavaScript, Arrays are predefined objects)
- You can use a general function to make arrays:
  - function Makearray(n) {
    - this.length = n;
    - for (var x = 0; x< n; x++) {
      - this[x] = 0;
    - }
    - return this;
  - }
- And then use it:
  - var myArray = new MakeArray (3);
- You can also use objects to make “multi-dimension” arrays (arrays of arrays)