

Class Title: JavaScript 1 Lecture #: 3

Chapter 7 Array, Date, Math and Type Related Objects

Adapted from
JavaScript: The Complete Reference 2nd Edition
by
Thomas Powell & Fritz Schneider
© 2004 Thomas Powell, Fritz Schneider, McGraw-Hill

UCSD EXTENSION

PINT

Class Title: JavaScript 1 Lecture #: 3

Arrays

- Declaring arrays

```
var firstArray = new Array();  
var secondArray = new Array("red",3,"green");  
var thirdArray = new Array(5);
```
- Since JS 1.2 you can use array literals as well

```
var firstArray = [];  
var secondArray = ["red", 3, "green"];  
var thirdArray = [,,,];
```
- Sparse arrays

```
Var fourthArray = [.,35,,,16,,23,];
```

UCSD EXTENSION

PINT

Class Title: JavaScript 1 Lecture #: 3

Accessing Arrays

- Arrays are zero index based
- Access using []

```
var myArray = [1,51,68];  
var x = myArray[0]; // it's 1  
var x = myArray[2]; // 68  
var x = myArray[4]; // undefined
```
- Assignment is just as basic

```
myArray[3] = 125;
```
- You can create sparse arrays easily this way

```
myArray[200] = 3;
```

UCSD EXTENSION

PINT

Class Title: JavaScript 1 Lecture #: 3

Accessing Arrays Contd.

- Big nuance
 - As a reference type (composite type as well) Arrays will act differently than primitives during assignment

```
var firstarray = ["Mars", "Jupiter", "Saturn"];
var secondarray = firstarray;
secondarray[0] = "Neptune";
alert(firstarray); // it has been changed!!!
```
- The reason for this has to do with the way the objects are stored in memory. We saw this same action when passing composite types to functions

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Removing Array Elements

- You could use the **delete** operator to remove an item from an array
 - Sets the element to undefined but does not reduce the length value for the array

```
var myArray = [2,45,64,6,45,67,8];
alert(myArray.length);
delete myArray[2];
alert(myArray.length);
```
- We actually have to write a somewhat troublesome routine to remove an item from the center of an array and cover holes, though pulling from the start or end is relatively easily
 - Think about setting `myArray.length = myArray.length - 1;`

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Arrays as Stacks

- Stacks = last in, first out (LIFO) – food trays
 - **push()** puts element(s) at the end
 - **pop()** takes an element off the end

```
var mystack = []; // []
mystack.push("first"); // ["first"]
mystack.push(10, 20); // ["first", 10, 20]
mystack.pop(); // ["first", 10] Returns 20
mystack.push(2); // ["first", 10, 2]
mystack.pop(); // ["first", 10] Returns 2
mystack.pop(); // ["first"] Returns 10
mystack.pop(); // [] Returns "first"
```

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Arrays as queues

- Similar to push() and pop() we have
 - unshift()** – inserts arguments in order at the beginning of the array shifting elements to the right
 - shift()** – removes first element from array and shift
- Queues = first in, first out (FIFO) – a line
 - We can simulate a queue with arrays using **shift()** and **push()**

```

var queue = [];
myqueue.push("first", 10); // ["first", 10]
myqueue.shift();          // [10]      Returns "first"
myqueue.push(20);         // [10, 20]
myqueue.shift();          // [20]      Returns 10
myqueue.shift();          // []        Returns 20

```

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Manipulating Arrays: Concat()

- Concat() – append items onto an array


```

var myArray = ["red", "green", "blue"];
alert(myArray.concat("cyan", "yellow"));

```
- Concat() does not modify in-place. You need to save it


```

var myArray = ["red", "green", "blue"];
myArray = myArray.concat("cyan", "yellow");

```
- Be careful when concatenating arrays into arrays you may find they are “flattened”

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Manipulating Arrays: Join()

- The **join()** method (JavaScript 1.1+) converts the array to a string and allows the programmer to specify how the elements are separated in the output.
- Typically, when you print an array, the output is a comma-separated list of the array elements. You can use **join()** to format the list separators as you'd like:


```

var myArray = ["red", "green", "blue"];
var stringVersion = myArray.join(" / ");
alert(stringVersion);

```
- Join does not change the array itself though

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Manipulating Arrays: Reverse()

- JavaScript 1.1+ also allows you to reverse the elements of the array in place. The **reverse()** method, as one might expect, reverses the elements of the array it is invoked on.


```
var myArray = ["red", "green", "blue"];
myArray.reverse();
alert(myArray);
```
- As you can see the array is affected by the method and does not have to be assigned

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Manipulating Arrays: Slice()

- The **slice()** method (JS 1.2+) returns a "slice" (subarray) of the array on which it is invoke
- Two arguments, the *start* and *end* index. If only one argument is given, the method returns the array of all items from that index to the end of the array.
 - start* and *end* can be negative and are interpreted as an offset from the end of the array **slice(-2)** returns an array holding the last two items of the array.

```
var myArray = [1, 2, 3, 4, 5];
myArray.slice(2); // returns [3, 4, 5]
myArray.slice(1, 3); // returns [2, 3]
myArray.slice(-3); // returns [3, 4, 5]
myArray.slice(-3, -1); // returns [3, 4]
myArray.slice(-4, 3); // returns [2, 3]
myArray.slice(3, 1); // returns []
```

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Manipulating Arrays: Splice()

- splice()** method (JS1.2+ /JScript 5.5+) can be used to add, replace, or remove elements of an array in place
- Syntax: **splice(start, deleteCount, replacevalues);**
 - where
 - start* is the index to perform the operation.
 - deleteCount* is the number of elements to delete

(If *deleteCount* is omitted, all elements from *start* to the end of the array are removed and returned)

Any further arguments represented by *replacevalues* (that are comma-separated, if more than one) passed to **splice()** are inserted in place of the deleted elements.

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Manipulating Arrays: Splice() Example

```
var myArray = [1, 2, 3, 4, 5];
myArray.splice(2, 2);
// myArray = [1, 2, 5]   Returned [3, 4]
myArray.splice(3);
// myArray = [1, 2, 3]   Returned [4, 5]
myArray.splice(2, 0, 15, 10);
// myArray = [1, 2, 15, 10, 3, 4, 5] Returns []
myArray.splice(1, 2, "red", 22, 7);
// myArray = [1, "red", 22, 7, 4, 5] Returns [2, 3]
```

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Manipulating Arrays: toString() & toSource()

- **toString()** returns a string containing the comma-separated values of the array
 - Invoked automatically during type conversion (e.g. when you print an array)
- **toSource()** creates a source representation of an array complete with brackets []
 - “[1,2,3]” versus “1,2,3”
 - With values returned from **toSource()** you can later use the **eval()** function to evaluate the string back into an array

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Sorting Arrays

- **Sort()** works similar to the **qsort()** function found in many standard programming libraries
- Sorts based on lexicographic order

```
var myArray = [14,52,3,14,45,36];
myArray.sort();
alert(myArray);
```

Note: In this case you will find 14 sorted before 3 as 1 sorts lexicographically higher than 3
- Possible to add your own sort function that when given two items returns -1 if first less than second item, 0 if same and 1 if first passed item is greater than second

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Sorting Modification

```
function myCompare(x, y)
{
  if (x < y)
    return -1;
  else if (x === y)
    return 0;
  else
    return 1;
}
```

Then we could use the function in the previous example like so

```
var myArray = [14,52,3,14,45,36];
myArray.sort(myCompare);
alert(myArray);
```

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Multidimensional Arrays

- You can create an array of arrays “multidimensional array”
`var tableOfValues = [[2, 5, 7], [3, 1, 4], [6, 8, 9]];`
- You would use a dual index method to access the array items
`alert(tableOfValues[2][2]); // returns 9`

UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Extending Arrays with Object Prototypes

- Since they are objects themselves you can extend the facilities provided using prototype
- Consider our new printing function

```
function myDisplay()
{
  if (this.length != 0)
    alert(this.toString());
  else
    alert("The array is empty");
}
Array.prototype.display = myDisplay;
```


UCSD EXTENSION

Class Title: JavaScript 1 Lecture #: 3

Extending Arrays Contd.

- We could then use the new method like so

```
var myArray = [4,5,7,32];  
myArray.display();  
// displays the array values
```
- ```
var myArray2 = [];
myArray2.display();
// displays the string "The array is empty"
```
- The value of object prototypes with arrays is that it allows us to add features to JS versions lacking certain built-in Array methods (e.g. push() )



---

---

---

---

---

---

---

---


Class Title: JavaScript 1 Lecture #: 3

## Boolean Type Object

- Boolean** is the built-in object corresponding to the primitive Boolean data type
- Use constructors like any other object  

```
var boolData = new Boolean (true);
```
- Without a value it sets as false  

```
var myBool = new Boolean();
```
- You can use **toString( )** and **toSource( )** on such an object
- The **typeof** statement will identify **Boolean** objects correctly as objects and you should be able to even add props to the object!



---

---

---

---

---

---


---

---

Class Title: JavaScript 1 Lecture #: 3

## Date

- The **Date** object provides properties and methods to manipulate dates and time.
- JavaScript stores date information internally as the number of milliseconds since “epoch” January 1, 1970 GMT
  - Beware of dates pre-epoch in some JS implementations
- You read the client’s date!
- Things are 0 enumerated (months are 0 – 11) and days are 0 – 6 from Sunday to Saturday. Of course months are enumerated from 1 - 31



---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Date Contd.

- Date() constructor is powerful

| Date() Argument              | Example                                           |
|------------------------------|---------------------------------------------------|
| None                         | var rightNow = new Date();                        |
| "month dd, yyy hh:mm:ss"     | var birthDay = new Date("March 24, 1970");        |
| Milliseconds                 | var someDate = new Date(79560003020);             |
| yyyy, mm, dd                 | var birthDay = new Date(1970, 2, 24);             |
| yyyy, mm, dd, hh, mm, ss     | var birthDay = new Date(1970, 2, 24, 15, 0, 0);   |
| yyyy, mm, dd, hh, mm, ss, ms | var birthDay = new Date(1970, 2, 24, 15, 0, 250); |

UCSD EXTENSION

---

---

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Manipulating Dates

- The **Date** object supports a variety of methods like **getMonth()**, **getDate()**, **setDate()**, **getMinutes()** and so on. Here are a few examples

```
var today = new Date();
document.write("The current date : "+today+"
");
document.write("Date.getDate() : "+today.getDate()+"
");
document.write("Date.getDay() : "+today.getDay()+"
");
document.write("Date.getFullYear() : "+today.getFullYear()+"
");
document.write("Date.getHours() : "+today.getHours()+"
");
document.write("Date.getMilliseconds() : "+today.getMilliseconds()+"
");
document.write("Date.getMinutes() : "+today.getMinutes()+"
");
document.write("Date.getMonth() : "+today.getMonth()+"
");
document.write("Date.getSeconds() : "+today.getSeconds()+"
");
document.write("Date.getTime() : "+today.getTime()+"
");
document.write("Date.getTimezoneOffset() : "+today.getTimezoneOffset()+"
");
document.write("Date.getYear() : "+today.getYear()+"
");
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Converting Dates to Strings

- You can convert a date to a string piece by piece and make your own format or just let type conversion run its course
- The **Date** object does support a variety of methods like **toString()**, **toGMTString()** and so on

```
var appointment = new Date("February 24, 1996 7:45");
document.write("toString():", appointment.toString());
document.write("
");
document.write("toUTCString():", appointment.toUTCString());
document.write("
");
document.write("toGMTString():", appointment.toGMTString());
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

---

---



Class Title: JavaScript 1 Lecture #: 3

## Converting Stings to Dates

- Use **Date.parse()** to parse a string and extract a date value (in ms) if possible
 

```
// Set value = December 14, 1982
var myDay = "12/14/82";
// convert it to milliseconds
var converted = Date.parse(myDay);
// create a new Date object
var myDate = new Date(converted);
// output the date
alert(myDate);
```
- If you pass a bad string you will get **NaN**

```
var myDay = "Friday, 2002";
var invalid = Date.parse(myDay);
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Date Gotchas

- The biggest gotcha of all that needs to be repeated over and over is that JavaScript dates rely on the user's local system clock which may be wrong!
- Other problems include:
  - Poor implementation of Date under Netscape 2
  - Problems with "extreme dates" pre 1AD and far in the future

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Global Object

- The **Global** object is a catchall for top-level properties and methods
- You can not create an instance of the global object
- It includes many of the "functions" that many people consider built-into JavaScript including:
  - escape()**, **unescape()**
  - eval()**
  - isFinite()**, **isNaN()**
  - parseFloat()**, **parseInt()**

UCSD EXTENSION

---

---

---

---

---

---

---

---


Class Title: JavaScript 1 Lecture #: 3

## Escape( ) and Unescape( )

- escape()** takes a string and returns a "URL safe" string where non-alphanumeric characters have been encoded in hex equivalents %xx. **unescape()** does the opposite translating an *x-url-encoded* string back to plain text

```
var aString="O'Neill & Sons";
// aString = "O'Neill & Sons"
aString = escape(aString);
// aString="O%27Neill%20%26%20Sons"
alert(unescape(aString));
// back to "O' Neill & Sons"
```

- The ECMA specification suggests using **encodeURIComponent()**, **decodeURI()**, and **decodeURIComponent()**, but so far few programmers seem to use them and not all browsers support the syntax

 PINT

---

---

---

---

---

---

---

---

---


---

Class Title: JavaScript 1 Lecture #: 3

## parseFloat( ) and parseInt( )

- parseFloat()** converts the string argument to a floating point number and returns the value. If the string cannot be converted, it returns **NaN**. The method should handle strings starting with numbers and peel off what it needs, but other mixed strings will not be converted. The method **parseInt( )** does the same thing except creates an integer

```
var x;
x = parseFloat("33.01568"); // x is 33.01568
x = parseFloat("47.6k-red-dog"); // x is 47.6
x = parseFloat("a567.34"); // x is NaN
x = parseFloat("won't work"); // x is NaN
x = parseInt("53"); // x is -53
x = parseInt("33.01568"); // x is 33
x = parseInt("47.6k-red-dog"); // x is 47
x = parseInt("a567.34"); // x is NaN
x = parseInt("won't work"); // x is NaN
```

 PINT

---

---

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3


## isFinite( ) and isNaN( )

- The method **isFinite( )** returns a Boolean indicating whether its number argument is finite.

```
var x;
x = isFinite("56"); // x is true
x = isFinite(Infinity) // x is false
```

- The method **isNaN( )** returns a Boolean indicating whether its argument is **NaN**

```
var x;
x = isNaN("56"); // x is False
x = isNaN(0/0); // x is true
x = isNaN(NaN); // x is true
```

 PINT

---

---

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Eval( )

- The **eval( )** function takes a string and executes it as JavaScript code.

```
var x;
var aString = "5+9";
x = aString; // x contains the string "5=9"
x = eval(aString); // x will contain the number 14
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Math Object Intro

- The **Math** object holds a set of constants and methods enabling more complex mathematical operations than the basic arithmetic operators discussed so far
- You cannot instantiate a **Math** object like you would an **Array** or **Date**. The **Math** object is static (automatically created by the interpreter) so its properties are accessed directly.
- Example, to compute the square root of **10**, the **sqrt()** method is accessed through the **Math** object directly:  

```
var root = Math.sqrt(10);
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Math Constants

- The **Math** object holds numerous constants

| Property     | Description                              |
|--------------|------------------------------------------|
| Math.E       | Base of natural log (Euler's constant e) |
| Math.LN2     | Natural log of 2                         |
| Math.LN10    | Natural log of 10                        |
| Math.LOG2E   | Log (base 2) of e                        |
| Math.LOG10E  | Log (base 10) of e                       |
| Math.PI      | PI (3.14.....)                           |
| Math.SQRT1_2 | Square root of 1/2                       |
| Math.SQRT2   | Square root of 2                         |

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Math Methods

| Method                     | Returns                                       |
|----------------------------|-----------------------------------------------|
| <b>Math.abs(arg)</b>       | <b>Absolute value of argument</b>             |
| Math.acos(arg)             | Arc-cosine of argument                        |
| Math.asin(arg)             | Arc sine of argument                          |
| Math.atan(arg)             | Arc tangent of argument                       |
| Math.atan2(y,x)            | Angle between x-axis and point x,y            |
| <b>Math.ceil(arg)</b>      | <b>Ceiling of argument</b>                    |
| Math.cos(arg)              | Cosine of argument                            |
| Math.exp(arg)              | E to arg power                                |
| Math.floor(arg)            | Floor of arg                                  |
| Math.log(arg)              | Natural log of arg                            |
| <b>Math.max(arg1,arg2)</b> | <b>The greater of arg1 and arg2</b>           |
| <b>Math.min(arg1,arg2)</b> | <b>The smaller of arg1 and arg2</b>           |
| <b>Math.random()</b>       | <b>A random number between 0 and 1</b>        |
| <b>Math.round(arg)</b>     | <b>Rounding to nearest integer up or down</b> |
| Math.sin(arg)              | The sine of the argument                      |
| Math.sqrt(arg)             | The square root of the argument               |
| Math.tan(arg)              | The tangent of the argument                   |

UCSD EXTENSION

---

---

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Using Random Numbers

- While many of the **Math** methods may never be used by some JS programmers, random can be very useful.
- The method returns a number from 0 to 1 so to get something in the range m to n inclusive use
 

```
Math.round(Math.random() * (n - m)) + m;
```

```
Math.round(Math.random() * (6-1))+1;
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Number Type Object

- Number** is the built-in object corresponding to primitive number data.
- The **Number()** constructor takes an optional argument specifying its initial value:
 

```
var x = new Number();
```

```
var y = new Number(17.5);
```
- Number** supports the constants
 

```
Number.MAX_VALUE, Number.MIN_VALUE,
```

```
Number.POSITIVE_INFINITY,
```

```
Number.NEGATIVE_INFINITY, Number.NaN
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## String Object

- **String** is the built-in object corresponding to the primitive string data type.
 

```
var s = new String();
var test = new String("Yes!");
```
- Like other data type objects you rarely use it this way however String does support many methods for string manipulation, examination, substring extraction, and even conversion of strings to marked up HTML.

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Strings Method Basics

- **Length** (number of characters in the string)
 

```
var s = "test";
alert(s.length); // 4
```
- **toUpperCase()** – return value of string to uppercase
 

```
var s = "abc";
alert(s.toUpperCase());
S = s.toUpperCase();
```
- **toLowerCase()** – returns string in lowercase. Works in similar manner as **toUpperCase()**;

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Examining Strings

- Individual characters can be examined with **charAt(arg)** where *arg* is the character index to look at in the string starting with 0
 

```
"JavaScript".charAt(1); // "a"
```
- You can retrieve the numeric value with **charCodeAt()**;
 

```
"JavaScript".charCodeAt(1); // returns 97
```
- Reverse direction is **fromCharCode()**

```
var aChar = String.fromCharCode(97); // a
var pet = String.fromCharCode(68,79,71); // DOG
```

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3


## Examining Strings

- **indexOf()** – takes a string argument and returns the index of the first occurrence of the argument or a value of -1 if not found  

```
“JavaScript”.indexOf(“Script”); // 4
```
- **lastIndexOf()** – works from the right hand side finding the last occurrence of a pattern  

```
“JavaScript”.indexOf(‘a’); // 1

“JavaScript”.lastIndexOf(‘a’); // 3
```
- You can also use the **match()** and **search()** methods to find patterns in strings using regular expressions




---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Manipulating Strings


- You can use **substring()** to slice out a piece of a string  

```
var x = “JavaScript”.substring(3) // x =

“aScript”
```
- You can also specify start and end index  

```
var x = “JavaScript”.substring(1,4) // x =

“avaS”
```
- The **slice()** method is an even more powerful method




---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3


## Manipulating Strings

- As previously seen we can use **+** to concatenate strings  

```
var newString = “This” + “ is ” + “ easy”;
```
- We can also use **concat()** method  

```
var s = “JavaScript”.concat(“ is “, “easy”);
```
- The **split()** method breaks up strings based upon a passed delimiter (often a space). The result is returned as an array  

```
var wordArray = “A simple example”.split(“ ”);
```




---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Strings and HTML Output

- Since we are outputting to HTML often we find String supports numerous special format as HTML methods.
- Examples
 

```

"test".bold() // test
"fun".fontcolor('red') // fun
Var x = "click"; document.write(x.link('http://www.yahoo.com'));
// writes click

```
- The complete list includes **anchor(name)**, **big()**, **blink()**, **bold()**, **fixed()**, **fontcolor(color)**, **fontsize(size)**, **italics()**, **link(location)**, **small()**, **strike()**, **sub()**, **sup()**
- String functions can be strung together
 

```

var x = test; alert(x.bold().italics().strike())

```

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Strings and HTML Output

- **String** methods to produce HTML should be replaced by DOM methods, but for now they are more well understood than the DOM
- HTML **String** methods generally produce non XHTML focused markup
- Some browsers may not support all HTML **String** methods like **blink()**

UCSD EXTENSION

---

---

---

---

---

---

---

---

Class Title: JavaScript 1 Lecture #: 3

## Summary

- All JavaScript primitive types have corresponding objects but you rarely directly use them
- The **String** and **Array** objects support some useful methods
- Some other built-in objects like **Math** and **Date** are also commonly used
- The **Global** object is a catchall for some useful functions such as **eval()** that we should be well aware of

UCSD EXTENSION

---

---

---

---

---

---

---

---