CodeByZ

first time right software

JavaScript Programiing

Zacky P.

February 2015

Content

Content

- What is JavaScript
- The Console
- Variables
- Operators
- Conditional
- Loops
- String
- Arrays
- Objects
- Functions
- Scope
- Data Structures

CodeByZ
first time right software

INTRO

CodeByZ
first time right software

WHAT IS JAVASCRIPT

What is JavaScript

- Full fledged programming language
- Most JS interpreters written in C/C++
- Historically used for client side (browser)
  - Interact with the user
  - Control the browser
  - Communicate asynchronously
  - Alter document content

- However…

What is JavaScript – Cont.

- Today also runs on server side
- Specifically Node.js*

* *ask me about my Node.js Course*

What is JavaScript – Cont.

- Compact yet very flexible
- Many tools written on top of core JS providing extra functionality:
  - API for Web Browsers
    - DOM, CSS, Web cams, 3D graphics, Canvas animation, audio, …
  - 3rd Party APIs
    - Twitter, Facebook, Google, …
  - 3rd Party Frameworks/Libraries

History

- Originally developed by **Brendan Eich** while working for Netscape

- Netscape also wanted a lightweight interpreted language that would appeal to nonprofessional programmers

History – Cont.

- Developed under the name “Mocha”
- Shipped Sept. 1995 as “LiveScript”
- Later renamed to JavaScript
  - Which caused a lot of confusion ever since
- Netscape also released server side JS
  - 2009 - JS for Netscape Enterprise Server
- Nov. 1996 - Netscape submitted JS to Ecma International
- June 1997 – First ECMAScript draft published

ECMAScript

- The scripting language standardized by Ecma International (association)
- Implementations:
  - **JScript** - Microsoft's dialect of the ECMAScript
    - *Because MS has to be “unique”...*
  - **ActionScript** - remember Flash?

    and of course…
  - **JavaScript**

Ecma Members

- ECMA Stands for... Guesses?

Ecma Members

- Google
- Adobe
- Microsoft
- HP
- Yahoo
- IBM
- Intel
- Apple
- Sony
- Toshiba
- Fujitsu
- Mozilla
- jQuery
- Wikimedia
- and many more….

ECMAScript - Versions

- ECMAScript 5
- Or ES5 for short
- December 2009

- ECMAScript Harmony (ES6)
- Expected circa Q4 2015

Characteristics

- Interpreted and not compiled
  - Execute each line as we come to it
- Syntax is similar to Java (hence the name)
  - However other than that they’re total strangers
- Case sensitive
  - MyVar and myVar are different
- Whitespace agnostic
  - Extra white spaces (more than one) are ignored
  - Not in strings of course

Characteristics – Cont.

- All statements must end with semicolon
  - If you don’t the interpreter will do it for you
  - But this is a bad habit
- Object oriented
  - var code = by.z;
- Executed as encountered in the flow

Characteristics – Cont.

- Dynamic
  - Arrays can be multi-typed

    ```
    var myArray = [‘my string’, false, 3.14159, null];
    ```

  - Variables can change type (“loosely typed”)

    ```
    var bob = 1;
    bob = “sacamano”;
    bob = {“name”: “sacamano”, “age”: 42};
    ```

JavaScript – NodeJS

- During the “Programming JavaScript” module we will use NodeJS as the execution environment
- It allow us to write simple application which reads and writes
- Don’t get use to it since we are going to switch to browser ASAP
  - See next slide

JavaScript – Browser

- This is our main focus
- Running and building applications that run under the browser
- Through our Academy we will execute JavaScript under the browser

CodeByZ
first time right software

THE CONSOLE

General

- The **console** object provides access to the browser's debugging console
- There is a *de facto* set of features that are typically provided
- There are extended features which depend on the browser vendor – not recommended
- 99% of the time we'll be logging stuff

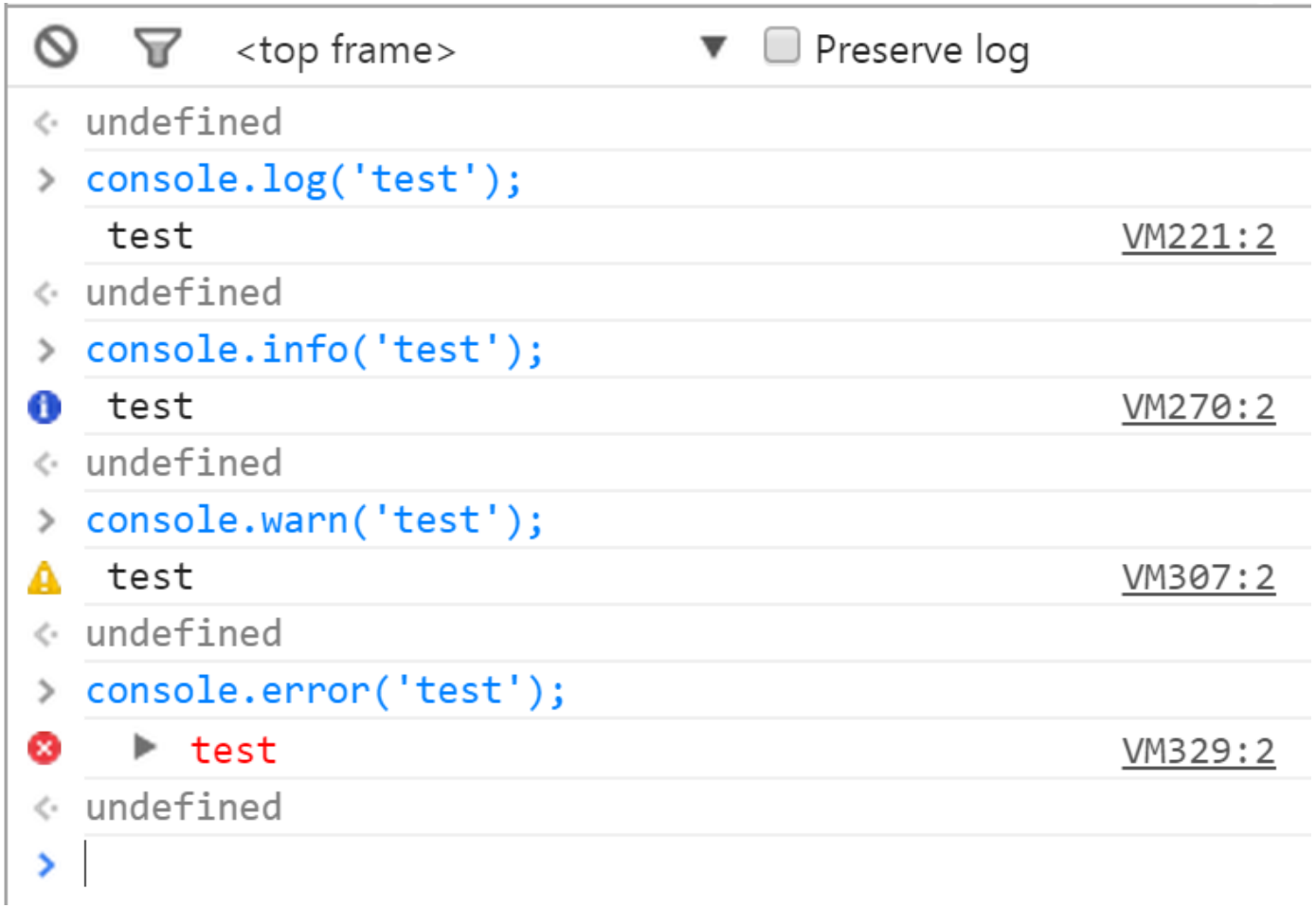
Outputting (logging)

- The most frequently-used feature
- Four categories:
  - console.log()
  - console.info()
  - console.warn()
  - console.error()

Outputting (logging) – Basics

```
console.log('hi there');
// hi there

var i = 42;
console.log('the value of i is ' + i);
// the value of i is 42

var someObject = { str: "some text", id: 5 };
console.log(someObject);
// Object {str: "some text", id: 5}
```

Outputting (logging) – Basics – Cont.



Outputting (logging) – Multiple Objects

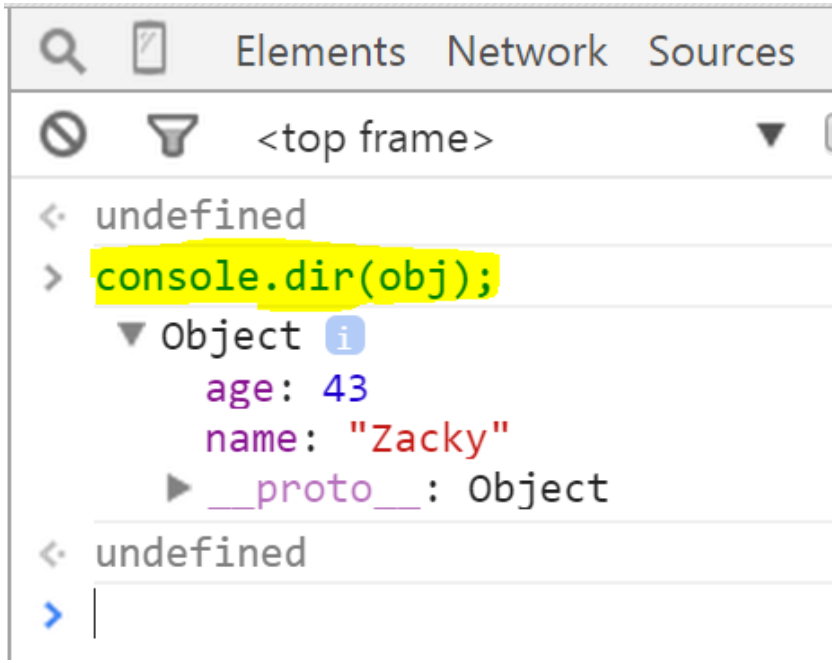
```
// outputting multiple objects

var car = "Dodge Charger";
var obj = {str:"Some text", id:5};
console.info("My first car was a", car, ". The object is: ", obj);

// My first car was a Dodge Charger. The object is:  Object {str:
    "Some text", id: 5}
```

Outputting (logging) – String Substitutes

```
// %o   – link to JS object – clicking expands object in explorer
// %d   – decimal / integer value
// %s   – string
// %f   – float point value

var     everything = "everything",
        answer = 42;

console.warn("The answer to %s is %d", everything, answer);
// prints: “The answer to everything is 42”
```

console.dir()

```
// similar to %o
// %o   – link to JS object – clicking expands object in explorer

var obj = {
    name: 'Zacky',
    age: 43
};

console.dir(obj);
```



Logging – Final Note

- It is advised never to write directly to console.log from your code
- Use a wrapper function / library instead
- IE-8 is for example notorious
- A wrapper could have fallback/polyfill

CodeByZ
first time right software

HANDS ON TIME – USE CONSOLE INSIDE NODEJS

CodeByZ
first time right software

VARIABLES

Variables

- Containers that can store values
- var keyword + variable name

```
var myVariable;
```

- After declaring a variable we can give it a value

```
myVariable = 'bob';
```

Variables – Cont.

- Retrieve variable's value by calling its name

```
myVariable;
```

- We can declare & assign in same line*

```
var myVariable = 'bob';
```

** Just be aware of hoisting – we'll talk about that*

Variables – Data Types

Variable	Explanation	Example
String	String of text. Enclosed in quotation marks	var myVar = "bob";
Number	A number. No quotation marks	var myVar = 42;
Boolean	A True/False value.	var isTruthy = true;
Array	Structuring allowing storage of multiple values in single reference	var myArr = [1, 'bob', 'Steve', 10]; var myVal = myArr[1]; // 'bob'
Object	Anything basically. Everything in JS is an object and can be stored in a variable.	var myVar = document.querySelector('h1');

undefined

- An unassigned variable is of type undefined
- A function returns undefined if a value was not returned
- A method or statement also returns undefined if the variable that is being evaluated does not have an assigned value

```
var x;
if (x === undefined) {
    // these statements execute
}
else {
    // these statements do not execute
}
```

null

- The value null is a JavaScript literal representing null or an "empty" value
- i.e. no object value is present

```
> var foo = null;
> foo
null
```

null vs. undefined

```
typeof null        // object (bug in ECMAScript,
                   // - should be null)
typeof undefined   // undefined
null === undefined // false
null  == undefined  // true
```

NaN and isNaN()

- The global **NaN** property is a value representing Not-A-Number

```
NaN === NaN;        // false
Number.NaN === NaN; // false
isNaN(NaN);         // true
isNaN(Number.NaN);  // true
```

Comments

```
// This is a comment

/*
This is a multi line
comment
right here
*/
```

CodeByZ
first time right software

OPERATORS

Operators

- Mathematical symbols
- Acting on two values or variables
- Producing a result

Operators – Cont.

Operator	Explanation	Symbol(s)	Example
Add, Concatenate	Add numbers or glue strings together	+	40 + 2; “CodeBy” + “Z”;
Subtract, Multiply, Divide	Do what you’d expect them to	-, *, /	50 – 8; 21 * 2; 84 / 2;
Assignment	Assigns a value to a variable	=	var myVar = “bob”;
Identity	Tests to see if two values equal. Returns a boolean	===	var myVar = 42; myVar === 42;
Negation, Not equal	Logical NOT, not identical	!, !==	var myVar = 42; myVar !== 41;

Assignment

```
var Code = 10;
var By = 10;
var Z = 22;

// is similar to
var Code = 10, By = 10, Z = 22;
```

typeof

- Returns a string indicating the type of the unevaluated operand
- The *typeof* operator is followed by its operand

```
// all the following return true
typeof 37 === 'number';
typeof "bla" === 'string';
typeof false === 'boolean';
typeof blabla === 'undefined'; // an undefined variable
typeof [1, 2, 4] === 'object'; // array is an object
typeof {a:1} === 'object';
typeof null === 'object'; // yup, null is also an object
```

typeof – Possible Return Values

Type	Result
undefined	"undefined"
null	"object" (see below)
Boolean	"boolean"
Number	"number"
String	"string"
Function object	"function"
Any other object	"object"

Increment / Decrement

```
var i = 1;
var j = ++i;  // pre-increment:  j equals 2; i equals 2
var k = i++;  // post-increment: k equals 2; i equals 3
```

Addition vs. Concatenation

```
var foo = 1;
var bar = '2';

var result = foo + bar; // result is now '12‘ – uh oh

console.log(foo + bar);  // 12
```

Forcing a String to Act as Number

```
var foo = 1;
var bar = '2';

// coerce the string to a number
console.log(foo + Number(bar)); // 3. Better
```

- Note, that above we called the Number constructor.
- We could do the same using the Unary plus operator

```
console.log(foo + +bar); // 3
```

Parantheses Indicate Precedence

```
2 * 3 + 5;        // returns 11; multiplication happens first
2 * (3 + 5);      // returns 16; addition happens first
```

Logical

- || → Logical OR
  - Stops evaluation when becomes true
- && → Logical AND
  - Stops evaluation when becomes false

- Notes
  - These are not bitwise operators
  - They return the *value* of the last operand evaluated
  - They do not return a Boolean!

Logical

```
var foo = 1, bar = 0, baz = 2;

foo || bar;   // returns 1, which is true
bar || foo;   // returns 1, which is true

foo && bar;   // returns 0, which is false
foo && baz;   // returns 2, which is true
baz && foo;   // returns 1, which is true
```

Logical – Cont.

- It is common to use logical operands for control flow:

```
// do something with foo if foo is truthy
foo && doSomething(foo);

// set bar to baz if baz is truthy,
// otherwise, set it to the return
// value of createBar()
var bar = baz || createBar();
```

Comparison

- Test whether values are equivalent or whether values are identical

```
var foo = 1, bar = 0, baz = '1', bim = 2;

foo == bar;    // returns false
foo != bar;    // returns true
foo == baz;    // returns true; careful! (coercion)

foo === baz;                 // returns false
foo !== baz;                 // returns true
foo === parseInt(baz);       // returns true

foo > bim;    // returns false
bim > baz;    // returns true
foo <= baz;   // returns true
```

Arithmetic Operators

Operator	Explanation	Example
% (mudulus)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
++ (Increment)	Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4.
-- (Decrement)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then, sets x to 2.

Arithmetic Operators – Cont.

Operator	Explanation	Example
- (Unary negation)	Unary operator. Returns the negation of its operand.	If x is 3, then -x returns -3.
+ (Unary plus)	The unary plus operator precedes its operand and evaluates to its operand but attempts to converts it into a number, if it isn't already	+3 // 3 +"3" // 3 +true // 1 +false // 0 +null // 0

=== and ==

- == means “equality with type coercion”
- === means “equality without type coercion”
- Means that values must equal in type as well

```
0 == false          // true
0 === false         // false, because they are of a different type
1 == "1"            // true, auto type coercion
1 === "1"           // false, because they are of a different type
'0' == false        // true , auto type coercion
'0' === false       // false
null == undefined           // true , auto type coercion
null === undefined          // false
```

Bitwise

Operator	Usage	Description
AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones
OR	a | b	eturns a one in each bit position for which the corresponding bits of either or both operands are ones.
XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
NOT	~ a	Inverts the bits of its operand.
Left Shift	a << b	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right.
Sign Propagation Right Shift	a >> b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off.
Zero Fill Right Shift	a >>> b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off, and shifting in zeroes from the left.

parseInt()

- Parses a string argument
- Returns an integer of the specified radix (base)
- Default radix is 10 (base 10)

```
parseInt(" 0xF", 16);       // 15
parseInt("       F", 16);   // 15
parseInt("Hello", 8);       // NaN
parseInt("0e0", 16);        // 224
```

CodeByZ
first time right software

CONDITIONAL CODE

Conditionals

- Code structures allowing expression testing
- And run different code depending on result
- The most common is the famous “if else”

```
var myStr = “Sweet fancy Moses”;

If (myStr === “Sweet fancy Moses”) {
    console.info(“Fancy”);
} else {
     console.info(“Not fancy”);
}
```

if else

```
var foo = true;
var bar = false;

if (bar) {
  console.log('hello!'); // this code will never run
}

if (bar) {
   // this code won't run
} else {
   if (foo) {
      // this code will run
   } else {
      // this code would run if foo and bar were both false
   }
}
```

Truthy and Falsy Things

- To use flow control successfully, it’s important to understand which kinds of values are “truthy” and “falsy”
- Sometimes, values that seem like they should evaluate one way actually evaluate another.

Values that Evaluate to true

```
true
'0'
'any string'
[]  // an empty array
{}  // an empty object
1   // any non-zero number
```

Values that Evaluate to false

```
false
0
''  // an empty string
NaN // JavaScript's "not-a-number"
null
undefined
```

Conditional w/ Ternary Operator

```
// set foo to 1 if bar is true otherwise 0
var foo = bar ? 1 : 0;

// conditional function invocation
var stop = false, age = 16;
age > 18 ? location.assign("continue.html") :
        stop = true;
```

Switch Statement

```
switch (foo) {
    case 'bar':
        alert('the value was bar -- yay!');
        break;

    case 'baz':
        alert('boo baz');
        break;

    default:
        alert('everything else is just ok');
        break;
}
```

CodeBy2
first time right software

LOOPS

for Loop

```
// logs 'try 0', 'try 1', ..., 'try 4'
for (var i=0; i<5; i++) {
    console.log('try ' + i);
}
```

for Loop – Cont.

for ([initialisation]; [conditional]; [iteration])
[loopBody]

- The ***initialisation*** statement
  - Executed only once, before the loop starts
  - Gives opportunity to prepare/declare variables

for Loop – Cont.

- The ***conditional*** statement
  - Executed before each iteration
  - Its return value decides whether the loop continues
- The ***iteration*** statement
  - Executed at the end of each iteration
  - Gives opportunity to change the state of important variables
  - Typically, this will involve incrementing or decrementing a counter

for Loop – Cont.

- The ***loopBody*** statement is what runs on every iteration
  - Can contain anything we want
  - Typically consists of multiple statements that execute
    - and so will wrap them in a block ({...}).

while Loop

```
var i = 0;
while (i < 100) {
    // This block will be executed 100 times
    console.log('Currently at ' + i);
    i++; // increment i
}
```

while Loop – Cont.

```
var i = 0;
while (++i < 100) {
    // This block will be executed 100 times
    console.log('Currently at ' + i);
}
```

do while Loop

```
do {
    // Even though the condition evaluates to false
    // this loop's body will still execute once.
    alert('Hi there!');
} while (false);
```

CodeByZ
first time right software

STRING

string

- Zero indexed array like objects
- Useful for holding text data
- Most-used operations on strings:
  - Check length with *length* property
  - Construct & concatenate using *+ , +=* operators
  - Find substrings location using *indexOf()*
  - Extracting substrings using *substring()*

string – Construction

- Directly as literal
  - 'You just blew my mind' // string
  - "You just blew my mind" // string
  - “In Thai: คุณเพียงแค่ พัด ใจของฉัน“
    - // Just make sure your is UTF-8 encoded
    - // and that your HTML contains a utf-8 meta tag (<meta charset="utf-8">)
- Using the String global object
  - String('You just blew my mind') // string
  - new String('You just blew my mind') // object

string – primitive vs. String object

- JS distinguishes between String objects and primitive string values
- Primitive:
  - Literals: denoted by double or single quotes
  - String('.....')
- String object:
  - **new** String('.....');

string – primitive vs. String object – Cont.

- JS automatically converts primitives to String objects
- So that we can use String object methods for primitive strings
- Developers rarely need to worry about String objects and 99.9% of time just use literal/primitive strings.
- Convert a String object to its primitive type using the *valueOf* method.

string – primitive vs. String object – Cont.

```
var myStr = new String('You just blew my mind');

console.log('myStr is an ' + typeof myStr);
// myStr is an object

console.log('myStr\'s value is: ' + myStr.valueOf());
// myStr's value is: You just blew my mind

console.log(myStr);
// String {0: "Y", 1: "o", 2: "u", 3: " ", 4: "j", 5: "u", 6: "s", 7: "t", 8: " ", 9: "b",
    10: "l", 11: "e", 12: "w", 13: " ", 14: "m", 15: "y", 16: " ", 17: "m", 18: "i",
    19: "n", 20: "d", length: 21, [[PrimitiveValue]]: "You just blew my mind"}
```

string – Special Characters

Code	Output
\0	the NUL character
\'	single quote escaping
\"	double quote escaping
****	backslash
\n	new line - advance downward to the next line
\r	carriage return – return to beginning of line
\v	vertical tab - used to speed up printer vertical movement (you're unlikely to ever use this)
\t	tab
\b	backspace
\f	form feed - advance downward to the next "page"
\uXXXX	Unicode representation of a character

string – Common Methods & Operators

Code	Explanation	Example

charAt()	Returns character at index.
	Not writeable!
		'And yada yada yada'.charAt(1); // returns “n“
		
		var str = ‘serenity now!’;
		Str[0] = ‘S’; // does not work

[]	Same as charAt	
+	Concatenation	var combined = ‘one two’ + ‘ ‘ + “three”;
+=	Concatenation	var combined = ‘Uno Dos ‘; combined += ‘ Tres’;
substring()	extracts characters from indexA up to but not including indexB.	'Wow! That\'s a lot of potatoes'.substring(0,4); // "Wow!”
trim()	removes whitespace from both ends of a string	" foo ".trim(); // “foo”

string – Common Methods & Operators 2

Code	Explanation	Example
toLowerCase()	returns the calling string value converted to lowercase	console.log('ALPHABET'.toLowerCase()); // 'alphabet'
toUpperCase()	returns the calling string value converted to uppercase	console.log('alphabet'.toUpperCase()); // 'ALPHABET'
replace()	returns a new string with some or all matches of a pattern replaced by a replacement *str*.replace(*regexp*|*substr*, *newSubStr*|*function*[, *flags*])	'Wow! That\'s a lot of potatoes'.replace('potatoes', 'tomatoes'); // "Wow! That's a lot of tomatoes"

string – Common Methods & Operators 3

Code	Explanation	Example

slice()	extracts a section of a string and returns a new string
	
	str.slice(*beginSlice*[, *endSlice*])
		var str1 = 'The morning is upon us.';
		var str2 = str1.slice(4, -2); console.log(str2);
		// morning is upon u

split()	splits a String object into an array of strings *str*.split([*separator*[, *limit*]])	*var friends = 'Jerry, George, Kramer, Elaine';* *friends.replace(' ', '').split(',');* *// ["Jerry", "George", " Kramer", " Elaine"]*

substr()	returns the characters in a string beginning at the specified location through the specified number of characters
	
	str.substr(*start*[, *length*])
		var str = 'abcdefghij';
		
		str.substr(1, 2); // bc
		str.substr(-3); // hij
		str.substr(-3, 2); // hi

CodeByZ
first time right software

ARRAYS

Array

- Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations
- Array elements are dynamic – can be of different types
- The array can dynamically grow/shrink
- Arrays are not guaranteed to be dense – can have “holes” in them
- The JavaScript **Array** global object is a constructor for arrays

Array – Cont.

- Construction options

```
var myArr = [1, 2, 'three', , , 'last']; // holes are undefined's
var myArr2 = new Array(1, 2, 'three', , , 'last');
var myArr3 = new Array(8); // 8 undefined's
```

- *Notice element types are dynamic and that arrays can be sparse (not dense)*

Array – Cont.

- Accessing Elements

```
var arr = [first element', 'second element'];
console.log(arr[0]);       // prints 'first element'
console.log(arr[1]);      // prints 'second element'
console.log(arr[arr.length - 1]);  // prints 'second element'
```

Array – Cont.

- Array elements are actually object properties
- However we can't access a property who's name is not valid

```
var arr = ['a', 'b', 'c'];

// syntax error just because 2 is
// not a valid property name
console.log(arr.2);

 // so we have to use this notation
console.log(arr[2]);  // prints 'c'
```

Array – Cont.

- This is the same as

```
// create an object with a property 3
var obj = { 3 : 'a',  'name' : 'ugly kid joe'};

console.log(obj.name);          // okay - prints 'ugly kid joe'
console.log(obj['name']);       // okay - prints 'ugly kid joe'
console.log(obj.3);             // not okay – SyntaxError
console.log(obj['3']);          // okay - prints 'a'
```

- *As you can see, an array is just an object*
- *Whose elements are object properties.*

Array – Cont.

- So if an array is an object with elements = object properties
- And object properties are accessed by string

```
console.log(obj['name']);
```

- Does that mean array indexes are strings too?

Array – Cont.

- Yes!

```
var years = [1950, 1960, 1970, 1980, 1990, 2000];
console.log(years[2]);      // 1970
console.log(years['2']);    // 1970 – works!
```

- *years[2] is coerced by the JS engine through implicit toString conversion and becomes years['2']*
- *Obviously it isn't necessary to use the object notation*

Array – Cont.

- Array **length** property
- Returns number of elements in array including sparse (undefined's)

```
var fruits = [];
fruits.push('banana', 'apple', 'peach');
console.log(fruits.length); // 3

// increasing array's length
fruits.length = 10; // adds 7 undefined's

// setting shorter length actually deletes elements
fruits.length = 2;
console.log(fruits);  // ["banana", "apple"]
```

Array – join, push, pop

```
// joining an array to string and splitting to array
var myArray = [ 'h', 'e', 'l', 'l', 'o' ];
var myString = myArray.join('');   // 'hello'
var mySplit = myString.split('');   // [ 'h', 'e', 'l', 'l', 'o' ]

// using as stack – push and pop
myArray = ['Code', 'By'];

// pushes new element at end of array.
// returns length of array = 3
myArray.push('Z'); // array now contains ['Code', 'By', 'Z']

// pops last element from end of array
// returns popped element = 'Z'
myArray.pop('Z'); // array now contains ['Code', 'By'] again
```

Array – splice

- The **splice()** method changes the content of an array
- By removing existing elements
- And/or adding new elements

array.splice(*start*, *deleteCount*[, *item1*[, *item2*[, ...]]])

Array – splice – Cont.

```
var myFish = ['angel', 'clown', 'mandarin', 'surgeon'];

// removes 0 elements from index 2, and inserts 'drum'
var removed = myFish.splice(2, 0, 'drum');
// myFish is ['angel', 'clown', 'drum', 'mandarin', 'surgeon']
// removed is [], no elements removed

// removes 1 element from index 3
removed = myFish.splice(3, 1);
// myFish is ['angel', 'clown', 'drum', 'surgeon']
// removed is ['mandarin']
```

Array – splice – Cont.

```
// reminder: myFish is ['angel', 'clown', 'drum', 'surgeon']

// removes 2 elements from index 0
// and inserts 'parrot', 'anemone' and 'blue'
removed = myFish.splice(0, 2, 'parrot', 'anemone', 'blue');
// myFish is ['parrot', 'anemone', 'blue', 'trumpet', 'surgeon']
// removed is ['angel', 'clown']
```

Array – Other Methods

- Array's prototype contains many more methods
- Read the docs to find more
  - **forEach** - executes a provided function once per element
  - **concat** – concatenates two arrays
  - **filter** – returns a list filtered by filter function/criteria
  - **shift** – removes first element from the array
  - **unshift** - adds one or more elements to beginning of array
  - **slice** – returns shallow copy of portion of array
  - **every** – tests if all elements pass a test (criteria callback)
  - **some** - tests if any element passes a test (criteria callback)
  - …

Array – Other Methods

- Array's prototype contains many more methods
- Read the docs to find more
  - **forEach** - executes a provided function once per element
  - **concat** – concatenates two arrays
  - **filter** – returns a list filtered by filter function/criteria
  - **shift** – removes first element from the array
  - **unshift** - adds one or more elements to beginning of array
  - **slice** – returns shallow copy of portion of array
  - **every** – tests if all elements pass a test (criteria callback)
  - **some** - tests if any element passes a test (criteria callback)
  - …

CodeBy2
first time right software

OBJECTS

Object Literals

- A comma-separated list of name-value pairs
- Wrapped in curly braces

```
// declaration
var myObject = {
    someString: 'some string value',
    numProps: 2,
    isTrue: false
};

// later on…
myObject.numProps; // 2
```

Object Literals – Cont.

- Properties can be of any data type
  - Arrays, functions
  - Nested object literals

```
var myOblLiteral = {
   // an array literal
   images: ["smile.gif", "grim.gif", "frown.gif", "bomb.gif"],
   pos: { // nested object literal
      x: 40,
      y: 300
   },
   onSwap: function() { // function
      // code here
   }
};
```

Object Literals – Syntax

- A colon separates property name from value

  propName: “value”

- A comma separates name-value pairs

  propName1: “value1”, // note the comma
  propName2: 42

- Should be no comma after last name-value pair

  propName1: “value1”,
  propName2: 42, // note the bad comma here

Object Literals – Iterating

- Use a *for* statement

```
var myObject = { a: 2, b:3 }

// Iterate the properties. Prints: a b
for (prop in myObject) {
        console.log(prop);
}

// Iterate the values. Prints: 2 3
for (prop in myObject) {
        console.log(myObject [prop]);
}
```

Object Literals – When To Use?

- When do we use it?
  - Function parameters
    - Allows high degree of flexibility
    - Don’t care about parameter order
    - Can update obj properties from within function body (as it is passed by ref)
  - Group data together
    - Minimizes using of globals and global scope
    - Encapsulates related data in single place

Object Literals – Drawback

- Can easily break due to bad syntax
  - Missing colons
  - Colon after the last name-value pair
- Especially when heavily nested
- Causes code to stop working

delete

- The delete operator removes a property from an object

delete *object.property*

**delete *object*['*property*']

delete – Examples

```
x = 42;             // creates the property x on the global object

var y = 43;         // creates the property y on the global object,
                    // and marks it as non-configurable

// x is a property of the global object and can be deleted
delete x;      // returns true

// y is not configurable, so it cannot be deleted
delete y;      // returns false

// delete doesn't affect certain predefined properties
delete Math.PI;   // returns false
```

delete – Examples Cont.

```
myobj = {
  h: 4,
  k: 5
};

// user-defined properties can be deleted
delete myobj.h; // returns true

// myobj is a property of the global object, not a variable,  so can be deleted
delete myobj;   // returns true

function f() {
       var z = 44;

     // delete doesn't affect local variable names
     delete z;     // returns false
}
```

CodeByZ
first time right software

FUNCTIONS

Functions

- Functions encapsulate reusable functionality
- Examples of built-in browser functions:

```
var myH1 = document.querySelector('h1');
alert("I am speachless. I am without speech");
```

- Defining our own functions

```
function multiply(num1, num2) {
    var result = num1 * num2;
    return result;
}
```

Declaration

```
// option 1 – defined at parse-time
function functionOne() { /*… */ };

// option 2 – defined at run-time
var functionTwo = function() { /*… */ };
```

Declaration – The Gotcha

```
// No error
functionOne();
function functionOne() { /* … */ };

// TypeError: undefined is not a function
functionTwo();
var functionTwo = function() { /* … */ };
```

Declaration – The Gotcha – Cont.

- But why? (clue: remember hoisting?)
- To fix - move run-time function declarations to top of scope

```
// everyone's happy now
var functionTwo = function() { /* ... */ };
functionTwo();
```

Callbacks / Functions as Arguments

- Functions are “first-class citizens”
- Can be assigned to variables or passed to other functions as arguments
- Very common and used frequently
- Ex: in jQuery, _underscore etc.

Callbacks / Functions as Arguments – Cont.

```
var myFn = function(fn) {
    var result = fn();
    console.log(result);
};

// Passing an anonymous function as an argument
myFn(function() {
    return 'hello world';
});  // logs 'hello world'
```

Callbacks / Functions as Arguments – Cont.

```
var myFn = function(fn) {
    var result = fn();
    console.log(result);
};

var myOtherFn = function() {
    return 'hello world';
};

// Passing a named function as an argument
myFn(myOtherFn); // logs 'hello world'
```

Naming Inline Functions

- Not required really but can be done
- We can name our inline functions/callbacks, which otherwise would be anonymous

```
// anonymous function callback function
// arguments.callee is the function itself. We use its name property
setTimeout(function() {
    console.log('My name is ' + arguments.callee.name);
    // My name is
}, 100);

// naming an inline callbacl function
setTimeout(function named () {
     console.log('My name is ' + arguments.callee.name);
    // My name is named
}, 100);
```

CodeByZ
first time right software

SCOPES

General

- Scopes refer to
  - **Where variables & functions are accessible**
  - **The context code is being executed in**

Global Scope

- When something is global means that it is accessible from anywhere in your code

```
// var accessible in the global scope
var monkey = "Gorilla";

// function accessible in the global scope
function greetVisitor () {
    return alert(“Scopes are important!");
}
```

- If that code was being run in a web browser, the function scope would be window

Local Scope

- Defined and accessible in a certain part of the code, like a function

```
function someFunc() {
    // brand is a function scope variable
    var brand = “Vandaley Industries";
    return alert(brand);
}

alert(brand); // Error – unavailable in global scope
```

Scope Inheritance

- Nested scopes have access to the containing scope's variables, functions, arguments

```
function saveName (firstName) {

    function capitalizeName () {
      // has access to containing function args (firstName)
      return firstName.toUpperCase();
    }
    var capitalized = capitalizeName();
    return capitalized;
}

alert(saveName("Crazy Joe Davola"));   // "CRAZY JOE DAVOLA"
```

Block vs. Function Scope

- For those of us coming from Java, C, C++, PHP:
- **JavaScript has function scope, not block scope!**

```
function f() {
    if (condition) {
        var tmp = ...;
        ...
    }
    // tmp still exists here!
}
```

CodeByZ
first time right software

DATA STRUCTURES

Static Array

- A fixed sized array
- Once created size does not change
- Supports random access
- Very common in modern languages (C#, Java)
  - JavaScript does not support it directly
  - Can simulate
- No reallocation since add/remove are not available

Dynamic Array (A.K.A ArrayList)

- An array that can change its size
- Supports add/remove
- Supports random access
- Might cause reallocation when adding/removing new items
- JavaScript supports that concept through the Array data type []

Stack

- Supports push & pop
- No random access (index based)
- A.K.A LIFO – Last in first out
- In JavaScript can be simulated using plain array

Linked List

- Supports insertFirst, insertLast, removeFirst, removeLast
- Each node is linked to the next node
- Sometimes is implemented as doubly linked list
- No random access ☹
- No reallocation when adding new node ☺

Binary Tree

- Supports add, remove
- Each node may have at most two children: left & right
  - Left child is smaller then parent
  - Right child is greater then parent
- Thus, data is always sorted
- No random access
- Efficient search O(logN)

Hash Table

- An array where items are located according to their hash value
- Two distinct items might have the same hash value
  - Will be linked as the same location
- The hash function should avoid duplicates as much as possible
- Very efficient searching almost O(1)

CodeBy2
first time right software

DEBUGGING

CodeByZ
first time right software

STYLE GUIDE

Coding Conventions

- Inspired by Sun's code conventions for Java
- Modified/adapted to JS
- Coding conventions are important for readability and maintainability

Indentation & Line Length

- **Indentation**
  - tabs = 4 spaces
- **Line length**
  - limit to 80 chars
  - Break when possible after an operator, preferably a comma
  - Indent next line by 8 spaces (2 tabs)

Comments & Variable Declarations

- **Comments**
  - Don’t state the obvious
  - Use humor not resentment
- **Variable Declarations**
  - Don’t use implicit/implied globals (not using the var keyword)
  - Minimize using of global variables
  - var statements should come first in function body
  - State each variable on own line with comment
  - Arrange vars alphabetically if possible

Variable Declarations - Example

Example:

```
function example(tableName) {
    var   currentEntry,   // currently selected table entry
          level,          // indentation level
          size;           // size of table
…
```

Function Declarations

- **Function Declarations**
  - No space between function name and left parentheses
  - One space before left curly brace
  - Body indented (4 spaces)
  - Right curly brace aligned with function declaration

```
function outer(c, d) {
    var e = c * d;

    function inner(a, b) {
        return (e * a) + b;
    }

    return inner(0, 1);
}
```

Function Declarations – Cont.

- **Function Declarations**
  - No space between function name and left parentheses
  - One space before left curly brace
  - Body indented (4 spaces)
  - Right curly brace aligned with function declaration

```
function outer(c, d) {
    var e = c * d;

    function inner(a, b) {
        return (e * a) + b;
    }

    return inner(0, 1);
}
```

Function Declarations – Cont.

- **Anonymous Function Declarations**
  - If a function literal is anonymous
  - There should be one space between the word **function** and the left parenthesis (
  - Otherwise it can appear that the function's name is ”function”

```
// one space after function keyword – clear that func is anonymous
div.onclick = function (e) {
    return false;
};

// no space here after function keyword - confusing
div.onclick = function(e) {
    return false;
};
```

Names

- Use only A .. Z, a .. z, 0 .. 9, _ (underscore)
- Variables and functions should start with a lower case letter
- Global variables should be all caps

```
I_AM_GLOBAL = "I am global";
```

- Constructor functions (that must be used with the ***new*** keyword) should start with a capital letter

```
function Person(name) { /* … */ }
```

Statements

- Each line should only contain one statement
- Each statement should end with a semicolon (;)
- A return statement should not use () around the return value;

Statements – if else

```
if (condition) {
    statements
} else if (condition) {
    statements
} else {
    statements
}
```

Statements – for

```
// for arrays and loops
for (initialization; condition; update) {
    statements
}

// for iterating an object's properties
for (property in object) {
    if (filter) {   // e.g: object.hasOwnProperty(property)
            statements
    }
}
```

Statements – while & do

```
while (condition) {
   statements
}

do {
   statements
} while (condition);
```

Statements – switch

```
switch (expression) {
   case expression:
      statements
      break;
   default:
      statements
      break;
}
```