COMP 2106 Advanced Server-Side Scripting with MEAN

Lesson 2 – Part 2 JavaScript Fundamentals Review

How to code JavaScript statements

A JavaScript ***statement*** has a syntax that's similar to the syntax of **Java**.

Whitespace refers to the **spaces**, **tab characters**, and **return characters** in the code, and it is **ignored** by the compiler.

- As a result, you can use spaces, tab characters, and return characters to format your code so it's easier to read.

In some cases, **JavaScript** will try to correct what it thinks is a missing semicolon by adding a semicolon at the end of a split line.

How to code JavaScript statements (continued)

The basic syntax rules

JavaScript is case-sensitive.

JavaScript ignores extra **whitespace** within statements.

Each **JavaScript** statement ends with a **semicolon**.

How to split a statement over two or more lines

Split a statement after:

- an arithmetic or relational operator such as **+**,**-**,*****,**/**,**=**,**==**,**>**, or **<**
- an opening brace (**{**), bracket (**[**), or **parenthesis**
- a closing brace (**}**)

Do not split a statement after:

- an **identifier**, a **value**, or the ***return*** keyword
- a closing bracket (**]**) or **parenthesis**

How to create identifiers

Identifiers are the names given to **variables**, **functions**, **objects**, **properties**, and **methods**.

In ***camel casing***, all of the words within an identifier except the first word start with capital letters.

Rules for creating identifiers

Identifiers can only contain **letters**, **numbers**, the **underscore**, and the **dollar sign**.

Identifiers **can't** start with a number.

Identifiers are **case-sensitive**.

Identifiers can be **any length**.

Identifiers **can't be** the same as ***reserved words***.

Avoid using **global properties** and **methods** as identifiers. If you use one of them, you won't be able to use the global property or method with the same name.

How to create identifiers (continued)

Valid identifiers in JavaScript:

subtotal

taxRate

index_1

calculate_click

$log

Camel casing versus underscore notation:

taxRate	tax_rate
calculateClick	calculate_click
emailAddress	email_address

Naming recommendations

Use meaningful names for identifiers. That way, your identifiers aren't likely to be reserved words or global properties.

Be consistent: Either use **camel casing** (taxRate) or **underscores** (tax_rate) to identify the words within the variables in your scripts.

If you're using underscore notation, use **lowercase** for all letters.

Please Note: camel casing is the conventional standard for Node.js developers

Reserved Words

abstract	else	instanceof	super
boolean	enum	int	switch
break	export	interface	synchronized
byte	extends	let	this
case	false	long	throw
catch	final	native	throws
char	finally	new	transient
class	float	null	true
const	for	package	try
continue	function	private	typeof
debugger	goto	protected	var
default	if	public	void
delete	implements	return	volatile
do	import	short	while
double	in	static	with
			yield

How to use objects, methods, and properties

An *object* has ***methods*** that perform functions that are related to the object as well as ***properties*** that represent the data or attributes that are associated with the object.

When you ***call*** a method, you may need to pass one or more ***parameters*** to it by coding them within the parentheses after the method name, separated by commas.

The ***window*** *object* is the ***global object*** for client-side JavaScript, and JavaScript lets you omit the object name and *dot operator* (period) when referring to the window object.

The Syntax for calling a method of an object

objectName.methodName (parameters)

The primitive data types

The number data type

The ***number data type*** is used to represent an **integer** or a **decimal** value that can start with a **positive** or **negative sign**.

An ***integer*** is a whole number.

A ***decimal value*** can have one or more decimal positions to the right of the decimal point.

If a result is stored in a **number data type** that is larger or smaller than the data type can store, it will be stored as the value Infinity or -Infinity.

The Boolean data type

The ***Boolean data type*** is used to represent a ***Boolean value***. A Boolean value can be used to represent data that has two possible states: true or false.

The primitive data types (continued)

Floating point numbers

In **JavaScript**, decimal values are stored as ***floating-point numbers***. In that format, a number consists of a **positive** or **negative sign**, one or more significant digits, an optional decimal point, optional decimal digits, and an optional exponent.

Unless you're developing an application that requires the use of very large or very small numbers, you won't have to use floating-point notation to express numbers.

The string data type

The ***string data type*** represents character *(**string**)* data. A string is surrounded by double quotes or single quotes. The string must start and end with the same type of quotation mark.

An ***empty string*** is a string that contains no characters. It is entered by typing two quotation marks with nothing between them.

How to code numeric expressions

To code a ***numeric expression***, you can use the ***arithmetic operators*** to operate on two or more values.

The ***modulus operator*** returns the remainder of a division operation.

An arithmetic expression is evaluated based on the ***order of precedence*** of the operators.

To override the order of precedence, you can use **parentheses**.

Because the use of increment and decrement operators can be confusing, the recommendation is that you only use these operators in expressions that consist of just a variable name followed by the operator.

How to code numeric expressions (continued)

Common arithmetic operators

Operator	Description	Example	Result
+	Addition	5 + 7	12
-	Subtraction	5 - 12	-7
*	Multiplication	6 * 7	42
/	Division	13 / 4	3.25
%	Modulus	13 % 4	1
++	Increment	counter++	adds 1 to counter
--	Decrement	counter--	subtracts 1 from counter

The order of precedence for arithmetic expressions

Order	Operators	Direction	Description
1	++	Left to Right	Increment Operator
2	--	Left to Right	Decrement Operator
3	* / %	Left to Right	Multiplication, Division, Modulus
4	+ -	Left to Right	Addition, Subtraction

How to work with numeric variables

A ***variable*** stores a value that can change as the program executes.

To ***declare*** a variable, code the keyword ***var*** and a variable name.

- To declare more than one variable in a single statement, code ***var*** and the variable names separated by commas.

Within an expression, a ***numeric literal*** is a valid **integer** or **decimal** number that isn't enclosed in quotation marks.

To **assign** a value to a variable, you use an ***assignment statement*** that consists of the variable name, an ***assignment operator***, and an expression.

- When appropriate, you can declare a variable and assign a value to it in a single statement.
- If you use a plus sign in an expression and both values are numbers, JavaScript adds them
- If both values are strings, JavaScript **concatenates** them.
- If one value is a number and one is a string, JavaScript converts the number to a string and concatenates.

When you do some types of arithmetic operations with **decimal values**, the results aren't always precise (although they are extremely close) – this is because decimal values are stored internally as floating-point numbers.

Numeric Variables – Examples

```
var subtotal;                           // declares one variable
var investment, interestRate, years;    // declares three variables
```

```
var subtotal = 74.00;                   // subtotal = 74.00
var salesTax = subtotal * 1;            // salesTax = 7.4
```

```
var subtotal = 74.95;                   // subtotal = 74.95
subtotal += 20.00;                      // subtotal = 94.95
```

```
var counter = 1;                        // counter = 1
counter = counter + 1;                  // counter now = 2
counter += 1;                           // counter now = 3
counter++;                              // counter now = 4
```

```
var subtotal = 74.95;                   // subtotal = 7,.95
var salesTax = subtotal * .1;           // salesTax = 7.,95000000000001
```

How to work with string and Boolean variables

To assign values to string variables, you can use the **+** and **+=** operators, just as you use them with numeric variables.

To ***concatenate*** two or more strings, you can use the **+** operator.

Within an expression, a ***string literal*** is enclosed in quotation marks.

Escape sequences can be used to insert special characters within a string like a **return character** that starts a new line or a **quotation mark**.

If you use a **plus sign** in an expression and both values are strings, JavaScript concatenates them.

```
var firstName = "Thomas", lastName = "Vanguard";     // assigns two string values
var fullName = lastName + ", " + firstName;          // fullName is Thomas Vanguard
```

```
var months = 120;
message = "Months: ";
message += months;                                   // message is "Months: 120"
```

```
var message = "A valid variable name\ncannot start with a number.";
var message = "This isn\'t the right way to do this.";
```

Common Escape Sequences

Operators	Description
\n	Starts a new line in a string.
\"	Puts a double quotation mark in a string
\'	Puts a single quotation mark in a string
\b	Backspace
\0	Nul character.
\t	Horizontal tab.
\v	Vertical tab.
\\	Backslash

How to use the parseInt and parseFloat methods

The window object provides **parseInt** and **parseFloat** methods that let you convert string values to integer or decimal numbers.

When you use the prompt method or a text box to get numeric data that you're going to use in calculations, you need to use either the parseInt or parseFloat method to convert the string data to numeric data.

NaN is a value that means "Not a Number". It is returned by the **parseInt** and **parseFloat** methods when the value that's being parsed isn't a number.

When working with methods, you can embed one method in the parameter of another. This is sometimes referred to as **object chaining.**

Method	Description
`parseInt`	Converts the string that's passed to it to an **integer data type** and returns that value. If it can't convert the string to an **integer,** it returns **NaN.**
`parseFloat`	Converts the string that's passed to it to a **decimal data type** and returns that value. If it can't convert the string to a **decimal** value, it returns **NaN.**

How to code conditional expressions

A ***conditional expression*** uses the ***relational operators*** to compare the results of two expressions.

A ***compound conditional expression*** joins two or more conditional expressions using the ***logical operators***.

The **isNaN** method tests whether a string can be converted to a number. It returns **true** if the string is not a number and **false** if the string is a number.

Operator	Description	Examples
==	Equal	lastName == "Vanguard" testScore == 10
!-	Not Equal	firstName != "Ray" months != 10
<	Less Than	age < 18
<=	Less Than or Equal	investment <= 0
>	Greater Than	testScore > 100
>=	Greater Than or Equal	rate / 100 >= 0.1

The === operator

Widely used in Node apps, JavaScript also contains the === operator

How is this different from the == operator?

== checks if 2 values are equal

=== checks if 2 values are equal **as well as** if the values have the **same data type** (aka "equality WITHOUT type coercion")

How the logical operators work

Both tests with the **AND** operator must be **true** for the overall test to be true.

At least one test with the **OR** operator must be true for the overall test to be true.

The **NOT** operator switches the result of the expression to the other Boolean value.

- For example, if an expression is true, the NOT operator converts it to false.

To override the order of precedence when two or more logical operators are used in a conditional expression, you can use **parentheses**.

Operator	Description	Examples
!	NOT	!isNAN(age)
&&	AND	age > 17 && score < 70
||	OR	isNAN(rate) || rate < 0

How to code if statements

An ***if statement*** always has one ***if clause***. It can also have one or more ***else if clauses*** and one ***else clause*** at the end.

The statements in a clause are executed when its condition is true. Otherwise, control passes to the next clause. If none of the conditions in the preceding clauses are true, the statements in the else clause are executed.

If necessary, you can code one if statement within the if, else if, or else clause of another if statement. This is referred to as ***nesting if statements.***

The Syntax of the if statement

If (condition-1)

{ statements }

else if (condition-2)

{ statements }

else if (condition-n)

{ statements }

else

{ statements }

How to code while and do-while loops

The ***while statement*** creates a ***while loop*** that contains a block of code that is executed while its condition is true. This condition is tested at the **beginning of the loop**, and the loop is skipped if the condition is false.

The ***do-while statement*** creates a ***do-while*** loop that contains a block of code that is executed while its condition is true. However, its condition is tested at the **end of the loop** instead of the beginning, so the code in the loop will always be executed at least once.

The Syntax of a while loop

while (condition)

{ statements }

The Syntax of a do-while loop

do

{ statements }

while (condition);

How to code for loops

The ***for statement*** is used when you need to **increment** or **decrement** a counter that determines how many times the ***for loop*** is executed.

Within the parentheses of a for statement, you code an expression that **initializes** a ***counter*** (or *index)* variable, a conditional expression that determines when the loop ends, and an increment expression that indicates how the counter should be incremented or decremented each time through the loop.

The syntax of a *for* statement

for (counterInitialization; condition; incrementExpression)

{ statements }

How to work with objects, functions, and events

How to use Date and String objects

To create a **Date object** and assign it to a variable, use the syntax shown above. Then, you can use the **Date methods** with the variable.

When you declare a **string variable**, a String object is automatically created. Then, you can use the **String methods** with the variable.

A few methods of a Date object

Method	Description
`toDateString()`	Returns a string with the formatted date.
`getFullYear()`	Returns the four-digit year from the date.
`getDate()`	Returns the day of the month from the date.
`getMonth()`	Returns the month number from the date. The months are numbered starting with zero (e.g. January is 0 and December is 11).

A few methods of a String object

Method	Description
`indexof(search, position)`	Searches for the first occurrence of the search string starting at the position specified or zero if position is omitted. If found, it returns the position of the first character, counting from 0. If not found, it returns -1.
`substr(start, length)`	Returns the substring that starts at the specified position (counting from zero) and contains the specified number of characters.
`toLowerCase()`	Returns a new string with the letters converted to lowercase.
`toUpperCase()`	Returns a new string with the letters converted to uppercase.

Examples of the Date and String objects

```
var today = new Data();                    // creates Data object with currant data
alert ( today.toDateString() );            // displays Fri Mar 09 2012 on 3/9/2012
alert ( today.gatFullYear() );             // displays 2012
alert ( today.getData() );                 // displays 9
alert ( today.getMonth() );                // displays 2, not 3 for March
```

```
var name = "Thomas Vanguard";
var name Upper = name.toUpperCase();       // nameUpper = THOMAS VANGUARD
var nameLength = name.length;              // nameLength = 15
var index = name.indexOf(" ");             // index = 6
var firstname = name.substr(0, index);     // firstname = THOMAS
```

How to create and call an anonymous function

A ***function*** is a block of code that can be ***called*** by other statements in the program. When the function ends, the program continues with the statement after the calling statement.

A function can require that one or more ***parameters*** be passed to it when the function is called. In the calling statement, these parameters can also be referred to as ***arguments***.

To return a value to the statement that called it, a function uses a ***return statement***. When the return statement is executed, the function returns the specified value and ends.

An ***anonymous function*** is one that is coded as shown below. Technically, the function has no name, it is stored in a variable, and it is referred to by the variable name.

In the **JavaScript** for an application, an anonymous function must be coded before any statements that call it. Otherwise, an error will occur.

```
var variableName = function(parameters) {
    // statements that run when the function is executed
}
```

Function Examples

An anonymous function with no parameters that doesn't return a value

```
var showYear = function() (
var today = new Date();
alert( "The year is " + today.getFullYear() );
}
```

An anonymous function with one parameter that returns a DOM element

```
var $ = function (id) {
return document.getElementById(id);
}
```

An anonymous function with two parameters that returns a value

```
var calculateTax = function ( subtotal, taxRate ) {
var tax = subtotal * taxRate;
tax = parseFloat( tax.toFixed(2) );
return tax;
}
```

How to create and call a named function

A ***named function*** is one that is coded as shown above. This is just another way to code functions.

In contrast to an anonymous function, a named function doesn't have to be coded before any statements that call it.

The syntax for a named function

```
function functionName (parameters) {
// statements that run when the function is executed
}
```

A named function with no parameters that doesn't return a value

```
function() showYear {
var today = new Date();
alert( "The year is " + today.getFullYear() );
}
```

When and how to use local and global variables

The ***scope*** of a variable or function determines what code has access to it.

Variables that are created inside a function are ***local variables***, and local variables can only be referred to by the code within the function.

Variables created outside of functions are ***global variables***, and the code in all functions have access to all global variables.

If you forget to code the var keyword in a variable declaration, the JavaScript engine assumes that the variable is global. This can cause debugging problems.

In general, it's better to pass local variables from one function to another as parameters than it is to use global variables. That will make your code easier to understand with less chance for errors.

How to work with arrays

How to create an array and refer to its elements

An ***array*** can store one or more ***elements***. The ***length*** of an array is the number of elements in the array.

One way to create an array is to use the **new** keyword, the name of the object (Array), and a set of parentheses that contains a length parameter. If you don't specify a length, the array doesn't contain any elements. If you specify a length, each element is set to **undefined**.

The other way to create an array is to code a set of brackets with or without a list of elements.

To refer to the elements in an **array**, you use an ***index*** where 0 is the first element, 1 is the second element, and so on.

The syntax for creating an array

Using the new keyword with the Array object name

```
var arrayName = new Array(length);
```

Using the brackets literal

```
var arrayName = [];
```

How to add and delete array elements

One way to add an **element** to the end of an array is to use the **length property** as the index.

If you add an element at a **specific index** that isn't the next one in sequence, undefined elements are added to the array between the new element and the end of the original array.

Property	Description
`length`	The number of elements in an array.

Operator	Description
`length`	Deletes the contents of an element and sets the element to undefined, but doesn't remove the element from the array.

Array Examples

```
var numbers = [1, 2, 3, 4];              // array is 1, 2, 3, 4
numbers[numbers.length] = 5;             // array is 1, 2, 3, 4, 5
```

```
var numbers = [1, 2, 3, 4];              // array is 1, 2, 3, 4
numbers[6] = 7;                          // array is 1, 2, 3, 4, undefined, undefined, 7
```

```
var numbers = [1, 2, 3, 4];              // array is 1, 2, 3, 4
delete numbers[2];                       // array is 1, 2, undefined, 4
```

How to use for loops to work with arrays

When you use a **for loop** to work with an array, you can use the counter for the loop as the index for the array.

Code that puts the numbers 1 through 10 Into an array

```
var  numbers = [];
for (var i=0; i < 10; i++) {
  numbers[i] = i + 1;
};
```

Code that displays the numbers array created above

```
var numberString = "";
for (var i=0; i < numbers.length; i++) {
  numberString += numbers[i] + " ";
}
alert(numberString);
```

How to use for-in loops to work with arrays

You can use a ***for-in statement*** to create a ***for-in loop*** that accesses only those elements in an array that are defined.

The syntax of a for-In loop

```
for (var elementIndex in arrayName) {
    // statements that access the elements
}
```

A for-In loop that displays the numbers array In a message box

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var numbersString = " ";
for (var index in numbers) {
  numbersString += numbers[index] + " ";
}
alert(numbersString);
```

How to use the methods of an Array object

The **push** and **pop** methods are commonly used to add elements to and remove elements from the end of an array.

Methods	Description
`push(elements_list)`	Adds one or more elements to the end of the array, and returns the new length of the array.
`pop()`	Removes the last element in the array, decrements the length, and returns the element that it removed.
`unshift(elements_list)`	Adds one or more elements to the beginning of the array, and returns the new length of the array.
`shift()`	Removes the first element in the array, decrements the array length, and returns the element that it removed.
`join(separator)`	When no parameter is passed, this method converts all the elements of the array to strings and concatenates them separated by commas. To change the separator, you can pass this method a string literal.
`toString()`	Same as the join method without any parameter passed to it.

Array method examples

How to use the push and pop methods to add and remove elements

```
var names = ["Mike", "Anne", "Joel"];
names.push("Ray", "Tom");                    // names is Mike, Anne, Joel, Ray, Tom
var removedName = names.pop();               // removedName is Tom
alert(names.join());                         // displays Mike, Anne, Joel, Ray
```

How to use the unshift and shift methods to add and remove elements

```
var names = ["Mike", "Anne", "Joel"];
names.unshift("Ray", "Tom");                 // names is Ray, Tom, Mike, Anne, Joel
var removedName = names.shift();             // removedName is Ray
alert(names.join());                         // displays Tom, Mike, Anne, Joel
```

How to use the join and toString methods

```
var names = ["Mike", "Anne", "Joel", "Ray"];
alert(names.join());                         // displays Mike, Anne, Joel, Ray
alert(names.join(" "));                      // displays Mike,  Anne,  Joel,  Ray
alert(names.toString());                     // displays Mike, Anne, Joel, Ray
```