Seneca
SCHOOL OF INFORMATION AND
COMMUNICATIONS TECHNOLOGY

WEB322

Web Programming Tools and Frameworks

Schedule | Graded Work
Resources | Cyclic Guide | MyApps Instructions
Code examples

WEB322 Week 3 Notes

Object-Oriented JavaScript Review

Now that we have our development environment all set up and are comfortable making a simple web server (with Node.js & Express.js), we can start making some real progress with our web applications. However, before we can dive into the deeper topics, we need to review some of the advanced Object-oriented JavaScript topics that we first discussed in WEB222.

Creating Objects (Object Literal)

The most simple and straight-forward way to create an object in JavaScript is to use “Object Literal Notation” (sometimes referred to as “object initializer” notation). The syntax for creating an object using this notation is as follows:

```
var obj = { property_1:   value_1,
            property_2:   value_2,
            // ...,
            "property n": value_n }; // properties can also be defined as a string`
```

So, if we wanted to create an object with the following properties:

- **name** (string)
- **age** (number)
- **occupation** (string)

and methods…

- **setName** (“setter” to set a new value for the “name” property)
- **setAge** (“setter” to set a new value for the “age” property)
- **getName** (“getter” to get the current value of the “name” property)
- **getAge** (“getter” to get the current value of the “age” property)

using “Object Literal” notation, we would write the code:

```
var architect = {name: "Joe",
                  age: 34,
                  occupation: "Architect",
                  setName: function(newName){this.name = newName},
                  setAge: function(newAge){this.age = newAge},
                  getName: function(){return this.name},
                  getAge: function(){return this.age}
                };
```

and access the data (properties) and functions (methods) using the following code, ie:

```
console.log(architect.name); // "Joe"
// or
console.log(architect.getName()); // "Joe"
```

which creates a simple “architect” objet. Recall that we must use the **“this”** keyword whenever we refer to one of the properties of the object inside one of it’s methods. This is due to the fact that when a method is executed, “age” (for example) might already exist in the global scope, or within the scope of the function as a local variable. To be absolutely sure that we are referring to the correct “age” property of the current object, we must refer to the “execution context” - ie: the object that is actually making a call to this method. We know the object has an “age” property, so in order to be more specific about *which* age variable that we want to change, we leverage the keyword **this**. “this” will refer to the “execution context”, ie: the object that called the function! So, **“this.age”** can be read literally as **“the age property on this object”**, which is exactly the property that we wish to edit.

However, while “this” allows us to be specific with which **properties** that we refer to in our **methods**, it can lead to some confusing scenarios. For example, what if we added a new “outputNameDelay()” method to our architect object that writes the architect’s name to the console after 1 second (1000 milliseconds):

```
// ...
outputNameDelay: function(){
  setTimeout(function(){
    console.log(this.name);
  },1000);
}
// ...
architect.outputNameDelay(); // outputs undefined
```

Everything looks correct and we have made proper use of the “this”, however because the setTimeout function is not executed as a method of our architect object, we end up with “undefined” being output to the console. There are a number of fixes for this issue (most noteworthy is the new “arrow function” syntax - discussed below), however one common way is to introduce a local variable (often named “that”) into the current scope that **holds a reference to “this”**

```
// ...
outputNameDelay: function(){
    var that = this;
    setTimeout(function(){
    console.log(that.name);
    },1000);
}
// ...
architect.outputNameDelay(); // outputs "Joe"
```

Now, we aren’t using the “this” keyword from within the setTimeout() function, but rather “that” from our outputNameDelay function and everything works as it should! (ie, “that” points to architect, since it was the architect that invoked the outputNameDelay method).

Creating Objects (“class” keyword)

If we wish to create multiple objects of the same “type” (ie: that have the same properties and methods, but with different values), we can leverage the “class” and “new” keywords, ie:

```
class architect{

    name;
    age;
    occupation = "architect"; // default value of "architect" for
occupation

    constructor(setName = "", setAge = 0){ // handle missing parameters
        this.name = setName;
        this.age = setAge;
    }
    setName(newName){this.name = newName}

    setAge(newAge){this.age = newAge}

    getName(){return this.name;}

    getAge(){return this.age;}

  }

// define new "architect objects using the "new" keyword with the
"architect" class

var architect1 = new architect("Joe", 34);
var architect2 = new architect("Mary", 49);

// samples of accessing properties and methods on both objects

console.log(architect1.name); // "Joe"

console.log(architect1.getName()); // "Joe"
console.log(architect2.getName()); // "Mary"
```

Notice how we specify the properties (with default values), a “constructor” function to take initialization parameters, as well as specify all of the methods within the “class” block.

NOTE: By default, the properties are declared as “public”. To create “private” properties / methods, simply use the “#” prefix for their identifier (see: “Private class features” for more information).

Advanced JavaScript / ES6 Features

So far, we have learned quite a bit about JavaScript; from its dynamically typed variables, to complex custom / built-in Objects. However, for us to properly understand some of the examples in the upcoming weeks, we need to discuss a few advanced techniques as well as new syntax / methods from the ES6 (ECMAScript 6) standard.

“var” vs “let” vs “const”

As we know, JavaScript is a **dynamically typed language** and we declare our variables using the keyword **var**. However, when we use the “var” keyword, we’re actually creating our variables on the **function scope** (effectively allowing access to the variable outside the scope in which it was declared). Fortunately ES6 has introduced the let & const keywords to solve this problem. See the below table for a comparison of **var**,**let** & **const**

var

- Declares a variable, optionally initializing it to a value.
- The scope of a variable declared with var is its current execution context, which is either the enclosing function or, for variables declared outside any function, global.

```
for(var i =0; i < 5; i++){
  // ...
}

console.log(i); // 5
```

let

- Declares a block scope local variable, optionally initializing it to a value.
- The scope of a variable declared with “let” is limited to the block, statement, or expression on which it is used.

```
for(let j=0; j < 5; j++){
  // ...
}

console.log(j); // ReferenceError: j is not defined
```

const

- Declares an immutable block scope local variable, optionally initializing it to a value.
- The scope of a variable declared with “const” is limited to the block, statement, or expression on which it is used. However, the value of a variable declared with “const” cannot change through re-assignment and cannot be redeclared.

```
for(const k=0; k < 5; k++){ // TypeError: Assignment to constant variable.
  // ...
}

console.log(k);
```

As we can see from the above examples, **let** & **const** behave more like variable declarations in C / C++. While still being dynamically typed, they will respect the scope in which they are declared and cannot be referenced before they are declared.

Error / Exception handling

One of the most important aspects of writing any program is elegantly handling errors. It is important to never let your program suddenly crash or enter an unknown state due to an unanticipated error. Up until now we have seen numerous mechanisms in JavaScript to handle certain types of logical errors; for example the global isNaN() function is a way to elegantly respond to a situation in which a number was expected, but not returned:

```
let x = "twenty";

let y = parseInt(x);

if(isNaN(y)){
    console.log("x cannot be converted to a number");
}else{
    console.log("success! the numeric value of x is: " + y);
}
```

Similarly, we can use the global isFinite() function to handle a situation where division by zero has occurred:

```
let x = 30, y = 0;

let z = x / y;

if(isFinite(z)){
  console.log("success! " + x + "/" + y + "=" + z);
}else{
  console.log(x + " is not divisible by " + y);
}
```

However, while these functions are extremely useful for handling logical errors, they are not sophisticated enough to handle a situation that would completely break your code and cause the program to fail. For example, consider the following example that uses our new “const” keyword:

```
const PI = 3.14159;

console.log("trying to change PI!");

PI = 99;

console.log("Haha! PI is now: " + PI );
```

Here, we are trying to change the value of a constant: PI. If we try to run this short program in Node.js, the program will crash before we get a chance to see the string “Haha! PI is now: 99”, or even “Haha! PI is now: 3.14159”. There is no elegant recovery and we do not get to exit the program gracefully. This can be a huge problem if, for example we were working with a live connection to a service and an unexpected error occurred. Our program would crash and we would not be able to respond to the error by alerting the user and properly closing the connection. Fortunately, before our program crashes in such a way, Node.js will **“throw”** an **“Error”** object that we can intercept using the **“try…catch”** statement:

```
const PI = 3.14159;

console.log("trying to change PI!");

try{
  PI = 99;
}catch(ex){
  console.log("uh oh, an error occurred!");
}

console.log("Alas, it cannot be done, PI remains: " + PI);
```

If we execute the above code in Node.js we will find that our program doesn’t crash and that our string: “Alas, it cannot be done, PI remains: 3.14159” gets correctly logged to the terminal! Additionally, we can execute a specific block of code right when the error is encountered; in this case we output “uh oh, an error occurred!”. This is not very useful to help us debug the error, but it better than having the program crash and at least we know that an error did indeed occur. If we wish to obtain additional information about the error, we can make use of some of the properties / methods of the **Error** object that was thrown as an exception and caught in our “catch” block. For example, we can alter the code to use the “message” property of the caught exception (ex) to display a more helpful error:

```
const PI = 3.14159;

console.log("trying to change PI!");

try{
  PI = 99;
}catch(ex){
  console.log("uh oh, an error occurred: " + ex.message);
  // outputs: uh oh, an error occurred: Assignment to constant variable.
}

console.log("Alas, it cannot be done, PI remains: " + PI);
```

By utilizing properties such as Error.message & Error.stack, we can gain further insight to exactly what went wrong and we can either refactor our code to remedy the error, or acknowledge that the error will happen and handle it gracefully.

Lastly, if we have some code that we would like to execute regardless of whether or not the code in our “try” block is successful, we can use a “finally” block:

```
const PI = 3.14159;

console.log("trying to change PI!");

try{
    PI = 99;
}catch(ex){
    console.log("uh oh, an error occurred: " + ex.message);
    // outputs: uh oh, an error occurred: Assignment to constant variable.
}finally{
    console.log("always execute code in this block");
}

console.log("Alas, it cannot be done, PI remains: " + PI);
```

Throwing Errors

Now that we know how to correctly handle errors that have been thrown by the Node.js runtime environment or by other code / modules included in our solutions, why don’t we try throwing our **own exceptions**? This is very straightforward and only requires the use of the **“throw”** keyword and (typically) an **Error** Object:

```
function divide(x,y){
    if(y == 0){
    throw new Error("Division by Zero!");
    }
    return x / y;
}

let a = 3, b = 0, c;

try{
    c = divide(a,b);
}catch(ex){
    console.log("uh oh, an error occurred: " + ex.message);
    // outputs: uh oh, an error occurred: Division by Zero!
    c = NaN;
}
```

```
console.log(a + " / " + b + " = " + c); // 3 / 0 = NaN
```

Notice how the code below the "throw" statement does not get executed, and the flow of execution goes directly into the catch block. This prevents the error from propagating and ensures that it is handled immediately. As you can see, we can throw a new error whenever we detect that an error *may* occur anywhere in our code. In the above example, we check if our second parameter (y) is zero (0) and rather than trying to do the division, we immediately throw a custom error with the message "Division by Zero!". If the function call exists in a "try" block (as above), the execution of the code will immediately continue in the "catch" block and we mitigate the error by setting "c" to NaN.

Promises

So far, while learning JavaScript, we have seen a number of circumstances where "asynchronous" code is used. That is, once the code has been invoked, it does not block the main thread of execution while it's working. Once it's complete, an event is triggered (at an undetermined time) and we can write code to work with the result of the asynchronous operation. A classic example of this is a simple AJAX request using the HXMLHttpRequest object from the client side (web browser). Once we send() the request, code is executed that works outside of our main sequence of execution to establish the connection, make a request, etc. If we assign a function to the value of the XMLHttpRequest object's onreadystatechange property, we can execute some code at a later, undetermined time (maybe the request is to a particularly slow server) and handle the updated status of the request. The important thing to understand is that we can still execute code in a sequential fashion **after** we initiate the request!

To see this in action, we can invoke the global setTimeout function (as we did in our architect.outputNameDelay function) to create a situation in which the execution of code takes some time to complete, ie:

```
// output "A" after a random time between 0 & 3 seconds
function outputA(){
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    setTimeout(function(){
        console.log("A");
    },randomTime);
}

// output "B" after a random time between 0 & 3 seconds
function outputB(){
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    setTimeout(function(){
        console.log("B");
    },randomTime);
}
```

```
// output "C" after a random time between 0 & 3 seconds
function outputC(){
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    setTimeout(function(){
        console.log("C");
    },randomTime);
}

// invoke the functions in order

outputA();
outputB();
outputC();
```

In the above example, we can invoke the outputA() function (which will output the character "A" after a random delay between 0 & 3 seconds) and then immediately invoke the following "outputB()" and "outputC()" functions in order. Each function is said to be "non-blocking" because even though it will take some time to perform it's function (ie: output a letter to the browser), it does not stop the main flow of execution when it is invoked. Essentially, what we are doing is kickstarting 3 separate functions that will each output their value to the console after a random amount of time. When this example is executed, there is absolutely no way to know what order the functions will output their content to the browser - ie it could be "ACB", "BCA", "CAB", etc. However, what if that order was important? For example, what if one of the functions relies on the output from one of the other functions? If this were the case they would have to be executed in a specific order.

Resolve & Then

Fortunately, JavaScript has the notion of the **"Promise"** that can help us solve this type of situation. Put simply, a Promise object is used for asynchronous computations (like the situation in the example above) and represents a value which may be available now, or in the future, or never. Basically, what this means is that we can place our asynchronous code inside a Promise object as a function with specific parameters ("resolve" and "reject"). When our code is complete, we invoke the "resolve" function and if our code encounters an error, we can invoke the "reject" function. We can handle both of these situations later with the .then() method or (in the case of an error that we wish to handle) the .catch() method. To see how this concept is implemented in practice, consider the following addition to the outputA() method from above:

```
// output "A" after a random time between 0 & 3 seconds
function outputA(){
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    return new Promise(function(resolve, reject){ // place our code inside
a "Promise" function
```

```
        setTimeout(function(){
            console.log("A");
            resolve(); // call "resolve" because we have completed the
function successfully
        },randomTime);
    });
}

// call the outputA function and when it is "resolved", output a
confirmation to the console

outputA().then(function(){
    console.log("outputA resolved!");
});
```

Our “outputA()” function still behaves as it did before (outputs “A” to the console after a random period of time). However, our outputA() function now additionally returns a **new Promise** object that contains all of our asynchronous logic and its status. The container function for our logic always uses the two parameters mentioned above, ie: **resolve** and **reject**. By invoking the **resolve** method we are setting the promise into the fulfilled state, meaning that the operation completed successfully and the character “A” was successfully output to the browser. We can respond to this situation using the “then” function on the returned promise object to execute some code **after** the asynchronous operation is complete! This gives us a mechanism to react to asynchronous functions that have completed successfully so that we can perform additional tasks.

Adding Data

Now that we have the Promise structure in place and are able to **“resolve”** the Promise when it has completed it’s task and **“then”** execute another function using the returned Promise object (as above), we can begin to think about how to pass data from the asynchronous function to the “then” method. Fortunately, it only requires a little tweak to the above the above example to enable this functionality:

```
// output "A" after a random time between 0 & 3 seconds
function outputA(){
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    return new Promise(function(resolve, reject){ // place our code inside
a "Promise" function
        setTimeout(function(){
            console.log("A");
            resolve("outputA resolved!"); // call "resolve" because we have
completed the function successfully
        },randomTime);
    });
}

// call the outputA function and when it is "resolved", output a
confirmation to the console

outputA().then(function(data){
    console.log(data);
});
```

Notice how we are able to invoke the **resolve()** function with a single parameter that stores some data (in this case a string with the text "outputA resolved!"). This is typically where we would place our freshly returned data from an asynchronous call to a web service / database, etc. The reason for this is that we will have access to it as the first parameter to the anonymous function declared inside the **.then** method and this is the perfect place to process the data.

Reject & Catch

It is not always safe to assume that our asynchronous calls will complete successfully. What if we're in the middle of an XHR (XMLHttpRequest) request and our connection is dropped or a database connection fails? To ensure that we handle this type of scenario gracefully, we can invoke the "reject" method instead of the "resolve" method and provide a reason why our asynchronous operation failed. This causes the promise to be in a "rejected" state and the ".catch" function will be invoked, where we can gracefully handle the error. The typical syntax for handling both "then" and "catch" in a Promise is as follows:

```
// output "A" after a random time between 0 & 3 seconds
function outputA(){
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    return new Promise(function(resolve, reject){ // place our code inside
a "Promise" function
        setTimeout(function(){
            console.log("-");
            reject("outputA rejected!"); // call "reject" because the
function encountered an error
        },randomTime);
    });
}

// call the outputA function and when it is "resolved" or "rejected, output
a confirmation to the console

outputA()
.then(function(data){
    console.log(data);
})
.catch(function(reason){
    console.log(reason);
});
```

Chaining Promises

As we have seen, the Promise object and pattern for dealing with asynchronous code (of any kind) is extremely powerful. We are able to effectively process the result of executing an asynchronous block of code whether it completes successfully (using .resolve & .then) or fails / gives undesired results (using .reject & .catch). However, there is one last feature that we should discuss before moving on, ie: “chaining” promises. Recall, when we first began discussing promises we saw an example with 3 asynchronous functions (“outputA()”, “outputB()” and “outputC()”) that always completed in a different order even though they were always invoked in the same order. This could potentially cause problems if one function depended on another for data.

With promises, we can reliably detect when an asynchronous block of code completes, so why not use this to invoke a second (dependant) asynchronous function? This is the notion of “chaining” promises - executing one piece of asynchronous code after another and optionally passing data. For example, if we wish to ensure that “outputA()”, “outputB()” and “outputC()” always execute in the same order, regardless of how long each task takes, we can update the code to use Promises in the following way:

```
// output "A" after a random time between 0 & 3 seconds
function outputA(){

    var randomTime = Math.floor(Math.random() * 3000) + 1;

    return new Promise(function(resolve, reject){
        setTimeout(function(){
            console.log("A");
            resolve("outputA() complete");
        },randomTime);
    });
}

// output "B" after a random time between 0 & 3 seconds
function outputB(msg){
    // NOTE: msg holds the 'resolve' message from the
    // previous function in the chain
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    return new Promise(function(resolve, reject){
        setTimeout(function(){
            console.log("B");
            resolve("outputB() complete");
        },randomTime);
    });
}

// output "C" after a random time between 0 & 3 seconds
function outputC(msg){
    // NOTE: msg holds the 'resolve' message from the
    // previous function in the chain
    var randomTime = Math.floor(Math.random() * 3000) + 1;

    return new Promise(function(resolve, reject){
        setTimeout(function(){
            console.log("C");
            resolve("outputC() complete");
        },randomTime);
    });
}

// invoke the functions in order

outputA()
.then(outputB)
.then(outputC)
.catch(function(rejectMsg){
    // catch any errors here
    console.log(rejectMsg);
});
```

Now, all three functions ("outputA()", "outputB()" & outputC()") have been updated to use promises and each return a new Promise object. Each promise is "resolved" once it's message has been written to the console – ie: "outputA()"'s promise is resolved once "A" is written to the, console, etc. We don't have to alter the functions to be aware of each other by passing in any related functions / callbacks and each function is treated as it's own isolated "promise" to output it's message to the browser.

The chaining actually occurs further down in the ".then()" method of each promise. Recall the ".then()" method of the promise accepts a function that is invoked once the promise is "resolved". So, we can first invoke the "outputA()" method, "then" when it is resolved, invoke the "outputB()" method. The trick that makes chaining work is that we must ensure the next function "in the chain", returns it's promise. We can continue this pattern to execute as many asynchronous functions (Promises) we like and be confident that they will always be executed in the order we invoke them.

NOTE: calling "resolve()" or "reject()" won't immediately exit the promise and invoke the related ".then()" or ".catch()" callback - it simply puts the promise in a "resolved" or "rejected" state and code immediately following the statement will still run, ie:

```
// ...
reject();
console.log("I will still be executed");
resolve(); // This promise will not be "resolved", since the resolve() call came after reject()
           // this also works the other way around.  A promise has been "settled" once reject or resolve has been called
// ...
```

If we want to immediately exit the function and prevent further execution of the code within the Promise, we can invoke the "return" statement, immediately following the "resolve()" or "reject()" call, ie:

```
// ...
reject(); return;
console.log("I will not be executed");
// ...
```

Arrow Functions

ES6 has introduced many new keywords, constructs, syntax and functionality to the JavaScript language (for a full list, refer back to the Compatibility Table). We cannot possibly discuss it all here, so we must concentrate on new syntax / functionality that is likely to be encountered when learning some of the frameworks in this course (ie: Node.js / Express.js, MongoDB, etc.).

One new concept that you will notice right away (or may have already noticed), is that there's a new operator: "=>" that we can use to declare anonymous functions – or "arrow functions":

```
var outputMessage = function(message){
  console.log(message);
};

// is the same as:

var outputMessageArrow = message => console.log(message);

// invoke each function to see the result

outputMessage("Function Expression");
outputMessageArrow("Arrow Function");
```

When we use the arrow (=>) syntax to create functions, we no longer need the “function” keyword and simple, one parameter / one line functions or methods can be greatly simplified as:

```
parameter => logic
```

However, if we have more than one parameter, or more than one line of logic, we can still use arrow functions to simplify the creation of anonymous functions by eliminating the “function” keyword:

```
var outputMessage = function(message1, message2) {
  console.log(message1);
  console.log(message2);
};

// is the same as:

var outputMessageArrow = (message1, message2) => {
  console.log(message1);
  console.log(message2);
};

// invoke each function to see the result

outputMessage("Function", "Expression");
outputMessageArrow("Arrow", "Function");
```

This still simplifies things from a syntax point of view, however both methods of declaring anonymous functions are still very similar. The syntax difference is most noticeable when we have simple functions that accept zero (0) parameters and perform a single line of logic, for example:

```
var outputMessage = function() {
  console.log("Hello Function Expression");
};

// is the same as:

var outputMessageArrow = () => console.log("Hello Arrow Function");

// invoke each function to see the result

outputMessage();
outputMessageArrow();
```

Implicit “return statement”

Arrow functions also implicitly return the value of the statement within the function. This can lead to very short, consice function declarations, ie:

```
var adder = function(num1, num2) {
    return num1 + num2
};

// is the same as:

var adderArrow = (num1, num2) => num1 + num2;

// invoke each function to see the result

console.log(adder(2, 2));
console.log(adderArrow(2,2));
```

Lexical “this”

Arrow functions are great for creating simplified code that is easier to read (sometimes referred to as “syntax sugar”), however there is another very useful and slightly misleading feature that we have yet to discuss: the notion of a “lexical ‘this’”. Recall that when we added the “outputNameDelay” method to the architect object, we had to overcome the issue with “this” pointing at the incorrect object by introducing a new local variable, “that”:

```
outputNameDelay: function(){
    var that = this;
    setTimeout(function(){
    console.log(that.name);
    },1000);
}
```

While this does solve the problem, wouldn’t it be better if we didn’t have to always create a new local variable to sit in for “this”? Fortunately, arrow functions actually use a “lexical this” instead of their own value for “this”, so functions defined using the arrow notation use the “this” value of their parent scope. This insures that if an arrow function is invoked in a different context than the one in which it is defined (like the above example), the value of “this” will not change.

Now, we can re-write the above function using an arrow function to achieve the same result without having to introduce any new variables to handle the “this” issue. Additionally, because it’s such a simple function, we can transform it into a single line:

```
outputNameDelay: function(){
    setTimeout(() => { console.log(this.name); }, 1000);
}
```

This is a typical use of arrow functions, ie to simplify a scenario in which we need to declare a function in place, often as a parameter to other functions. We don't have to concern ourselves with how "this" will behave in the new context and the added "syntax sugar" makes the operation much simpler to read and shorter to code.

A Word of Warning

Be careful when using arrow functions, as not every situation calls for a "lexical this". For example, when we declare methods on an object using object literal notation, we always want "this" to point to the current object, so "lexical this" doesn't make sense and arrow functions will actually fail to behave as expected:

```
var test1obj = {
  a: "a",
  b: () => console.log(this.a)
}

test1obj.b(); // undefined

var test2obj = {
  a: "a",
  b: function() { console.log(this.a); }
}

test2obj.b(); // "a"
```

In addition, arrow functions **do not** have any notion of the arguments object and also **cannot** be used as function constructors and will throw an error when using the new operator (ie: Function is not a constructor).

Sources

- MDN - Working with Objects
- JavaScript Reference