Seneca College

Web Programming

WEB322 – Week 3 – Class 1

WEB322 H

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 t 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 refer "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...

- **setAge** (simple "setter" to set a new value for the "age" property)
- **setName** (simple "setter" to set a new value for the "name" property)

using "Object Literal" notation, we would write the code:

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

which creates a simple "architect" objet. Recall that we must use the **"this"** keyword whenever we refer to one of the propertie

object inside one of it's methods. This is due to the fact that when a method is executed, "age" (for example) might already exi global scope, or within the scope of the function as a local variable. To be absolutely sure that we are referring to the correct " property of the current object, we must refer to the "execution context" – ie: the object that is actually making a call to this me know the object has an "age" property, so in order to be more specific about *which* age variable that we want to change, we le the keyword **this**. "this" will refer to the "execution context", ie: the object that called the function! So, **"this.age"** can be read lit **"the age property on this object"**, which is exactly the property that we wish to edit.

Now, if we want to create more objects with these same properties & methods, we can leverage JavaScripts native Object.crea method:

```
Object.create(proto[, propertiesObject])
```

This method will create a brand new object and use an existing object as it's **prototype** (explained further down). In practice, tl give the new object all of the properties, methods and values of the existing object while still being it's own, new instance. For example, if we wish to create two new *architect* objects, we can simply call **Object.create()** with our previous **architect** object a first parameter:

```
var architect1 = Object.create(architect);
var architect2 = Object.create(architect);
```

Now both **architect1** and **architect2** are new objects that have the same properties, methods and values as the original **archit** object. However, because they are each their own instance, we can change their properties and manipulate their data as singl entities:

```
architect2.setName("Mary");

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

Creating Objects (Function Closures)

Before we move on to one of the more powerful ways of creating objects in JavaScript (ie: Function Constructors), let's discuss practical application of the **closure** design pattern in JavaScript. Recall, that a **closure** is created whenever a function returns a function, or an object containing functions. This creates an interesting situation when executing the returned function – JavaSc actually executes it within the scope of the containing function, which means that the returned function has access to the loca variables of the function in which it was defined. To see this in action, consider the following simple example:

```
function counter(){
  var localCounter = 0; // declare "localCounter" within the "counter()" function scope

  return function(){ // return a new function that references "localCounter"
    localCounter++; // increment it by 1
    return localCounter; // return the value of "localcounter"
  };
```

```
}

// call the counter() function and get a reference to the new function
var count = counter();

// call the new function
console.log(count()); // outputs: 1
console.log(count()); // outputs: 2
console.log(count()); // outputs: 3
```

In this example, we declare a function **counter()** which serves no purpose except to declare a local variable **"localCounter"** (*Re* whenever a variable is declared using the **var** keyword, it exists on the global scope, **unless** it is declared within a function, in v case it is "private" and exists within the function's scope). The counter() function also defines a new (anonymous) function that increments the value of localCounter and returns it's value. The counter() function returns this function, just like it would retur other value. The catch is that when the returned function is invoked later, it is actually executed in the scope of the parent "cou function and therefore has access to localCounter, even though the function was invoked from a different scope. In terms of C Oriented programming, this is an approximation of the concept of Encapsulation.

Why don't we try to recreate our **architect** object using this idea:

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

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

console.log(architect1.name); // undefined

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

In this case, we create 3 "local" variables: **name**, **age** and **occupation**. Since we're using a function to do the work of ensuring encapsulation, we are able to initialize some of the local variables by passing their value into the function when it is first invok can be thought of as a **constructor** for our "object"). Also, instead of returning a single function that has access to the local var why don't we return a simple object that consists of **multiple** functions that have access to the variables? This gives us more fr to work with the private data in different ways.

Now, to create our **"architect1"** / **"architect2"** objects, we call the **architect()** function and pass in a couple of values as initializ our private variables. Once the **architect()** function executes, we will have a new object with 4 functions that have access to ou properties! To illustrate this point, if we try to access the "name" property directly, we are given a value of "undefined" because doesn't exist on the **architect1** object. However the **architect1** object does have access to it through the **getName()** function, b it exists in a closure and therefore is executed in the **architect** scope.

While this does a pretty tidy job of enforcing the idea of encapsulation and protecting our data from being accessed directly, it pose a significant problem from an Object-Oriented perspective. The "architect1" and "architect2" objects do not actually hold themselves! The are just simple objects, each containing 4 functions that happen to be executed within a scope that has acces some hidden data. So, while they are indeed objects, they are not objects in the sense that they hold both **data** *and* **functions** single component (object).

Creating Objects (Function Constructor)

One of the more advanced & powerful ways of creating complex objects in JavaScript is by using **"Function Constructors"** and "new" operator. Essentially, we can specify how instances of each "new" object will be created by writing a function that follows specific pattern – for example:

```
// Declare a function to initialize our "new" object with
// properties (ie: "objectProperty")
function myObjectInitializer(initialVal){
  this.objectProperty = initialVal;
}

// add methods (ie: "objectMethod") to the myObjectInitializer function prototype
myObjectInitializer.prototype.objectMethod = function(){ return this.objectProperty };

// create a new object and initialize the objectProperty with the value "Hello"
var myObject = new myObjectInitializer("Hello");

// execute the "objectMethod" on the new object
console.log(myObject.objectMethod()); // "Hello"
```

In the above example, we are using a function to define all of the properties of the object (later created using the "new" operat the same way that we declare properties in a "class" in C++. These properties (declared using the "this" keyword) will get addec new object once the "new" operator is used to create a new "instance". Additionally, because we are using a function to define object, we can leverage the function properties to initialize the new object with some values – in this case, we set "objectPrope "Hello".

We can define the methods of the new object in either the function (using this.functionName = function(){};) or on the prototy function (as in the above example). It is generally preferred to add the methods to the function prototype, since all new object created using this function constructor (ie: myObjectInitializer) will have access to it's prototype once they are created (using "r second added benefit is if we were to change this function later in the code, all of our objects would be updated to use the ne (since they're all referring to the method in the prototype).

To illustrate this concept, why don't we recreate our "architect" object using this method:

```
function architect(setName, setAge){
  this.name = setName;
  this.age = setAge;
  this.occupation = "architect";
}

architect.prototype.setName = function(newName){this.name = newName};
architect.prototype.setAge = function(newAge){this.age = newAge};
architect.prototype.getName = function(){return this.name};
architect.prototype.getAge = function(){return this.age};

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

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

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

This looks very similar to how we created our architects using the closure pattern, doesn't it? However, there's few key differen

- New "architect" objects (ie: "architect1" & "architect2") have their own **name**, **age**, & **occupation** properties
- New "architect" objects do not have any methods directly, however they all refer to the same prototype (architect.proto which contains all of the methods. These methods can work with the correct data for each new architect object, becaus are utilizing the "this" keyword.

"this" keyword

As we have seen, both primary ways of creating objects in JavaScript ("Object Literal" & "Function Constructor") make regular u "this" keyword. This is an important concept in JavaScript, so before we move on to Prototypal Inheritance, let's just do a quick

> **"this" always holds a reference to the "context" of the function (ie: the object actually invoking the function).**

So, when we declare an object with methods, we always make sure that each method refers to the properties in the object wit "this" keyword. This is because we wish to be specific about which property that we wish to reference and "this" always points object invoking the method. So, the **architect1.setName()** method will always work with the **architect1.name** property and sir the **architect2.setName()** method will always work with the architect2.name

While "this" allows us to be specific with which **properties** that we refer to in our **methods**, it can lead to some confusing scen For example, what if we added a new "outputNameDelay()" method to our architect object that writes the architect's name to t console after 1 second (1000 milliseconds):

```
// ...
architect.prototype.outputNameDelay = function(){
  setTimeout(function(){
    console.log(this.name);
  },1000);
}
```

```
// ...
architect2.outputNameDelay(); // outputs undefined
```

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

```
// ...
architect.prototype.outputNameDelay = function(){
  var that = this;
  setTimeout(function(){
    console.log(that.name);
  },1000);
};
// ...
architect2.outputNameDelay(); // outputs "Mary"
```

Now, we aren't using the "this" keyword from within the setTimeout() function, but rather "that" from our outputNameDelay fu and everything works as it should! (ie, "that" points to architect2, since it was the architect2 that invoked the outputNameDela method).

Prototypal Inheritance

Prototypal Inheritance is a very interesting and complex topic in JavaScript. There's a lot to learn about how it is implemented language, however for our purposes we will primarily concentrate on how it impacts our objects / object creation when using Function Constructor notation. For a full treatment of Objects & Prototypal inheritence, see: Introducing JavaScript objects fro "Learn web development" series.

So far, we have seen how to create our "architect" object using this notation. We actually made use of the "Prototype" property "architect" function to define the methods of our new architect objects (see above). Essentially, what is happening here is that refer to a Constructor Function's prototype (ie "architect.prototype"), we are really referring to another, separate object that all instances of "architect" (ie: "architect1" and "architect2") will reference via their own internal property "__proto__" (or "[[prototyp

So, why is so important for us? Well, when you make a call to a method or reference a property on any object, the JavaScript ru will actually check for their existence on the object's prototype as well as the object itself. Therefore, it can be said that "archite "architect2" **inherit** getName(), setName(), getAge() and setAge() from their prototype and any future properties or methods de on the prototype will be automatically picked up by each new / existing instance! This is easy to verify using the built in Object.getPrototypeof() function, for example:

```
// ...
console.log(architect2); // outputs: { name: 'Mary', age: 49, occupation: 'architect' }

console.log(Object.getPrototypeOf(architect2)); // outputs: { setName: [Function],
                                                //            setAge: [Function],
                                                //            getName: [Function],
                                                //            getAge: [Function] }
```

```
// ...
```

From the above code, it is clear that the "architect2" instance does not actually have it's **own** methods, but we can invoke them architect2 object and the JavaScript runtime will check its prototype for their existence and execute them as though they were actually happens often in JavaScript and is the reason that when we create a String (for example), we have access to properties .length or methods like .split(), .slice(), .substr(), etc. (see: String.prototype on MDN). We didn't have to specify each of those pr / methods, however we automatically **inherited them** from the global String Object's prototype.

To see why this concept is so powerful, why don't we add a new method to the architect prototype **after** we create our archite architect2 instances:

```
function architect(setName, setAge){
  this.name = setName;
  this.age = setAge;
  this.occupation = "architect";
}

architect.prototype.setName = function(newName){this.name = newName},
architect.prototype.setAge = function(newAge){this.age = newAge},
architect.prototype.getName = function(){return this.name},
architect.prototype.getAge = function(){return this.age}

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

architect.prototype.newMethod = function(){
  return "Hello: " + this.name;
};

console.log(architect2.newMethod()); // outputs: "Hello: Mary"
```

As you can see from above, we are able to add a new method (newMethod) to the architect prototype at any time and becaus architect instances (ie: architect2) use that prototype, they automatically get access to the method!

Advanced JavaScript / ES6 Features

So far, we have learned quite a bit about JavaScript; from how it handles simple and complex custom / built-in Objects to desig patterns like closures, modules, callback functions, etc. However, for us to properly understand some of the examples in the u weeks, we need to discuss a few advanced techniques as well as new syntax / methods from the new ES6 (ECMAScript 6) stanc important thing to note however, is that **ES6** is **still being implemented** across desktop & mobile browsers as well as JavaScrip runtimes. Most of what we will discuss will be understood by modern browsers and 100% of the topics below will be understo Node.js. However, it is a good idea to reference the following ES6 Compatibility Table if you are unsure whether your target br will fully understand the feature that you wish to use.

"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

the "var" keyword, we're actually creating our variables on the **global scope**. As we have seen, there's no way to prevent this ex use "var" within the **scope of a function**. This is why certain design patterns like IIFE (Immediately-Invoked Function Expressio often used to prevent variables from becoming global, ie:

```
// declare an anonymous function & immediately invoke it
(function () {
  var x = 5
  console.log(x) // 5
})();

console.log(x) // undefined
```

This is fairly concise and works well, but wouldn't it be better if we had more control over the variables without introducing ad programming constructs? Fortunately ES6 has introduced the let & const keywords to solve this problem. See the below table 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 func 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 us `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 However, the value of a variable declared with "const" cannot change through re-assignment and cannot b redeclared.

const

```
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 dynam 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 progra suddenly crash or enter an unknown state due to an unanticipated error. Up until now we have seen numerous mechanisms i JavaScript to handle certain types of logical errors; for example the global isNaN() function is a way to elegantly respond to a s 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 t 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 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 get to exit the program gracefully. This can be a huge problem if, for example we were working with a live connection to a serv an unexpected error occurred. Our program would crash and we would not be able to respond to the error by alerting the use properly closing the connection. Fortunately, before our program crashes in such a way, Node.js will **"throw"** an **"Error"** object 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 dor remains: 3.14159" gets correctly logged to the terminal! Additionally, we can execute a specific block of code right when the er encountered; in this case we output "uh oh, an error occurred!". This is not very useful to help us debug the error, but it bette having the program crash and at least we know that an error did indeed occur. If we wish to obtain additional information abc error, we can make use of some of the properties / methods of the **Error** object that was thrown as an exception and caught i "catch" block. For example, we can alter the code to use the "message" property of the caught exception (ex) to display a more 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 car 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 successfu 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 cod modules included in our solutions, why don't we try throwing our **own exceptions**? This is very straightforward and only requi 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 catcl This prevents the error from propagating and ensures that it is handled immediately. As you can see, we can throw a new erro whenever we detect that an error *may* occur anywhere in our code. In the above example, we check if our second parameter ((0) and rather than trying to do the division, we immediately throw a custom error with the message "Division by Zero!". If the call exists in a "try" block (as above), the execution of the code will immediately continue in the "catch" block and we mitigate 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 th has been invoked, it does not block the main thread of execution while it's working. Once it's complete, an event is triggered (a undetermined time) and we can write code to work with the result of the asynchronous operation. A classic example of this is AJAX request using the HXMLHttpRequest object from the client side (web browser). Once we send() the request, code is execu works outside of our main sequence of execution to establish the connection, make a request, etc. If we assign a function to th of the XMLHttpRequest object's onreadystatechange property, we can execute some code at a later, undetermined time (mayl request is to a particularly slow server) and handle the updated status of the request. The important thing to understand is th 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.prototype.outputNameDelay fur 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 seconds) and then immediately invoke the following "outputB()" and "outputC()" functions in order. Each function is said to be blocking" because even though it will take some time to perform it's function (ie: output a letter to the browser), it does not st main flow of execution when it is invoked. Essentially, what we are doing is kickstarting 3 separate functions that will each outp value to the console after a random amount of time. When this example is executed, there is absolutely no way to know what 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 cas 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 obje used for asynchronous computations (like the situation in the example above) and represents a value which may be available in the future, or never. Basically, what this means is that we can place our asynchronous code inside a Promise object as a fun 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 the case of an error that we wish to handle) the .catch() method. To see how this concept is implemented in practice, consider 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" functio
        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. Th container function for our logic always uses the two parameters mentioned above, ie: **resolve** and **reject**. By invoking the **reso**

method we are setting the promise into the fulfilled state, meaning that the operation completed successfully and the charact was successfully output to the browser. We can respond to this situation using the "then" function on the returned promise ob execute some code **after** the asynchronous operation is complete! This gives us a mechanism to react to asynchronous functi 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 **"th** execute another function using the returned Promise object (as above), we can begin to think about how to pass data from th asynchronous function to the "then" method. Fortunately, it only requires a little tweak to the above the above example to ena 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" functio
        setTimeout(function(){
            console.log("A");
            resolve("outputA resolved!"); // call "resolve" because we have completed the func
        },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 wit text "outputA resolved!"). This is typically where we would place our freshly returned data from an asynchronous call to a web database, etc. The reason for this is that we will have access to it as the first parameter to the anonymous function declared ir **.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 scenario gracefully, we can invoke the "reject" method instead of the "resolve" method and provide a reason why our asynchr operation failed. This causes the flow of execution to move into the ".catch" function, where we can gracefully handle the erro 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" functio
        setTimeout(function(){
            console.log("-");
            reject("outputA rejected!"); // call "reject" because the function encountered an
        },randomTime);
    });
}

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

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 a
to effectively process the result of executing an asynchronous block of code whether it completes successfully (using .resolve
or fails / gives undesired results (using .reject & .catch). However, there is one last feature that we should discuss before movi
"chaining" promises. Recall, when we first began discussing promises we saw an example with 3 asynchronous functions ("out
"outputB()" and "outputC()") that always completed in a different order even though they were always invoked in the same ord
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 afte
another and optionally passing data. For example, if we wish to ensure that "outputA()", "outputB()" and "outputC()" always exe
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("outputA() 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 Pro object. Each promise is "resolved" once it's message has been written to the console – ie: "outputA()"'s promise is resolved onc 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 a function that is invoked once the promise is "resolved". So, we can first invoke the "outputA()" method, "then" when it is reso invoke the "outputB()" method. The trick that makes chaining work is that we must ensure the next function "in the chain", ret promise. We can continue this pattern to execute as many asynchronous functions (Promises) we like and be confident that th always be executed in the order we invoke them.

Arrow Functions

ES6 has introduced many new keywords, constructs, syntax and functionality to the JavaScript language (for a full list, refer ba
Compatibility Table). We cannot possibly discuss it all here, so we must concentrate on new syntax / functionality that is likely
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 car
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
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
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 s
The syntax difference is most noticeable when we have simple functions that accept zero (0) parameters and perform a single
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();
```

Lexical "this"

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

```
architect.prototype.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 arr
notation use the "this" value of their parent scope. This insures that if an arrow function is invoked in a different context than
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 n
variables to handle the "this" issue. Additionally, because it's such a simple function, we can transform it into a single line:

```
architect.prototype.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 para
to other functions. We don't have to concern ourselves with how "this" will behave in the new context and the added "syntax s

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 object, we always want "this" to point to the current object, so "lexical this" doesn't make sense and arrow functions will actua 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 constructor will throw an error when using the new operator (ie: Function is not a constructor).
