Week 7 Notes

Further Functions

Functions are first-class objects

Call and Apply Methods

call() is used to set the value of **this** inside a function to an object that is provided as the first argument.

```
function sayHello(){

    return `Hello, my name is ${ this.name }`;

}

const clark = { name: 'Clark' };

const bruce = { name: 'Bruce' };


sayHello.call(clark);

<< 'Hello, my name is Clarke'


sayHello.call(bruce);

<< 'Hello, my name is Bruce'

function sayHello(greeting='Hello'){

    return `${ greeting }, my name is ${ this.name }`;

}


sayHello.call(clark, 'How do you do');

<< 'How do you do, my name is Clark'


sayHello.call(bruce);

<< 'Hello, my name is Bruce'
```

apply() works in the same way as **call()** but arguments of the function are provided as an array.

```
square.apply(null, [4])

<< 16
```

Custom Properties

You can add your own properties to functions the same way you add properties to an object:

square.description = 'Squares a number that is provided as an argument'

Memorization

Result caching- If a function takes some time to compute a value, we can save the result in a **chache** property.

```
function square(x){
    square.cache = square.cache || {};
    if (!square.cache[x]) {
        square.cache[x] = x*x;
    }
    return square.cache[x]
}
```

If we try calling the function a few times, we can see that the cache object stores the results:

```
Copy
square(3);
<< 9


square(-11);
<< 121


square.cache;
<< {"3": 9, "-11": 121}
```

Immediately Invoked Function Expressions (IIFE or "iffy")

An anonomus function that is invoked as soon as it is defined. You do this by adding parentheses at the end of the funciotn definition

```
(function(){

const temp = 'World';

console.log(`Hello ${temp}`);

})();

<< 'Hello World'
```

Temporary Variables

You can't remove a variable from the scope once declared. To avoid errors and confusion with a variable that is only needed temporarily, we can place any code that uses the temporary variable inside an IIFE so it is only available while the IIFE is invoked, then it will disappear.

```
let a = 1;

let b = 2;


(()=>{

   const temp = a;

   a = b;

   b = temp;

})();


a;

<< 2


b;

<< 1


console.log(temp);

<< Error: "temp is not defined"
```

Initialization Code

An IIFE can be used to set up any initialization code there will be no need for again. It will only run once.

```
(function() {

    const name = 'Peter Parker'; // This might be obtained from a cookie in reality

    const days = ['Sunday','Monday','Tuesday','Wednesday','Thursday', 'Friday','Saturday'];

    const date = new Date(),today = days[date.getDay()];

    console.log(`Welcome back ${name}. Today is ${today}`);


})();
```

<< 'Welcome back Peter Parker. Today is Tuesday'

Safe way to use Strict Mode

When coding, if you place 'use strict' at the beginning of a file is dangerous because if you are using other people's code, the might not have coded in strict mode.

The solution: Place all of your code in an IIFE to make sure only the code inside the IIFFE is forced to use strict mode.

```
(function() {

    'use strict';


// All your code would go inside this function


})();
```

Creating self-contained Code Blocks

Use an IIFE to enclose a block of code inside its own private scope so there is no interference with other parts of the program. Good for testing pupposes

```
(function() {

    // block A

    const name = 'Block A';
```

```
    console.log(`Hello from ${name}`);

    }());


    (function() {

    // block B

    const name = 'Block B';

    console.log(`Hello from ${name}`);

}());


<<  Hello from Block A

    Hello from Block B
```

Functions that Define and Rewrite Themselves

The way to allow a function to define and redefine itself is by assigning an anonymous function to a variable that has the same name as the function:

```
function party(){

    console.log('Wow this is amazing!');

    party = function(){

        console.log('Been there, got the T-Shirt');

    }

}
```

This logs a message in the console, then redefines itself to log a different message in the console. When the function has been called once, it will be as if it was defined like this:

```
function party() {

    console.log('Been there, got the T-Shirt');

}
```

Every time the function is called after the first time, it will log the message 'Been there, got the T-Shirt':

```
        party();

        << 'Wow this is amazing!'


        party();

        << 'Been there, got the T-Shirt'


        party();

        << 'Been there, got the T-Shirt'
```

If the function is also assigned to another variable, this variable will maintain the original function definition and not be rewritten.

If any properties have previously been set on the function, these will be lost when the function redefines itself

Init-Time Branching

Used with feature detection to create fumctions that rewrite themselves. Speeds up program.

We can define a function based on whether certain methods are supported.

```
function ride(){

  if (window.unicorn) {

    ride = function(){

    // some code that uses the brand new and sparkly unicorn methods

    return 'Riding on a unicorn is the best!';

    }

  } else {

    ride = function(){

    // some code that uses the older pony methods

    return 'Riding on a pony is still pretty good';

    }

  }
```

```
    return ride();

}

ride(); // the function rewrites itself, then calls itself

<< 'Riding on a pony is still pretty good'
```

Once the function has been invoked, it's rewritten based on the browser's capabilities. We can check this by inspecting the function without invoking it:

Copy

ride

```
<< function ride() {

    return 'Riding on a pony is still pretty good';

    }
```


Recursive Functions

This is a function that invokes itself until a condition is met. Useful in iterative processes.

```
function factorial(n) {

  if (n === 0) {

    return 1;

  } else {

    return n * factorial(n - 1);

  }

}
```

This function will return 1 if 0 is provided as an argument (0 factorial is 1), otherwise it will multiply the argument by the result of invoking itself with an argument of one less. The function will continue to invoke itself until finally the argument is 0 and 1 is returned. This will result in a multiplication of 1, 2, 3 and all the numbers up to the original argument.

Callbacks

Functions passed to other functions as arguments and then invoked inside the function they are passed to.

Event-driven Asynchronous Programming

Instead of waiting for an event to occura callback can be created and invoked when the event happens.The code is able to run asynchronously, or out of order.

Here's an example of a function called wait() that accepts a callback. To simulate an operation that takes some time to happen, we can use the setTimeout() function to call the callback after a given number of seconds:

```
function wait(message, callback, seconds){

   setTimeout(callback,seconds * 1000);

   console.log(message);

}
```

Now let's create a callback function to use:

```
function selfDestruct(){

   console.log('BOOOOM!');

}
```

If we invoke the wait() function then log a message to the console, we can see how JavaScript works asynchronously:

```
wait('This tape will self-destruct in five seconds ... ', selfDestruct, 5);

console.log('Hmmm, should I accept this mission or not ... ?');
```

<< 'This tape will self-destruct in five seconds ... '

<< 'Hmmm, should I accept this mission or not ... ? '

<< 'BOOOOM!'

Creating a Promise

Create a promise with a constructor function. It takes a function called an **executor** as an argument. The executor initializes the promise and starts the asynchronous operation. Accepts tow functions as arguments: **resolve()** (if operation is successful) and **reject()** (if operation fails)

```
const promise = new Promise( (resolve, reject) => {
```

```
   // initialization code goes here

   if (success) {

      resolve(value);

   } else {

      reject(error);

   }

});
```

```
const dice = {

   sides: 6,

   roll() {

      return Math.floor(this.sides * Math.random()) + 1;

   }

}
```

Now let's create a promise that uses the dice.roll() method as the asynchronous operation and considers rolling a 1 as a failure, and any other number as a success:

Copy

```
const promise = new Promise( (resolve,reject) => {

   const n = dice.roll();

   setTimeout(() => {

      (n > 1) ? resolve(n) : reject(n);

   }, n*1000);

});
```

This creates a variable called promise that holds a reference to the promise. The promise calls the roll() method and stores the return value in a variable called n. Next, we use an if-else block to specify the conditions for success (rolling any number higher than 1) and failure (rolling a 1). The setTimeout() method we met in Chapter 9 is used to add a short delay based on the number rolled. This is to mimic the time taken for an asynchronous operation to complete.

Dealing with a settled promise

Once a promise has been settled, use **then()** then method to deal with the outcome. The first argument is a fulfilment function that's called when the promise is resolved(data from resolve()). The second argument is a rejection function that is called if the promise is rejected (data from reject()).

promise.then(result => console.log(`Yes! I rolled a ${result}`), result => console.log(`Drat! ... I rolled a ${result}`));

Can also use **catch()** method to say what to do if operation fails:

promise.catch(result => console.log(`Drat! ... I rolled a ${result}`));

then() and catch() can be chained together:

promise.then(result => console.log(`I rolled a ${result}`))

    .catch(result => console.log(`Drat! ... I rolled a ${result}`));

Chaining multiple promises

login(userName)

.then(user => getPlayerInfo(user.id))

.then(info => loadGame(info))

.catch(throw error)

Async functions

Uses **async** keyword to allow you to write asynchronous code as if it was synchronous. Uses **await** operator. Wraps the return value of the function in a promise that can be assigned to a variable.

async function loadGame(userName) {

  try {

    const user = await login(userName);

    const info = await getPlayerInfo (user.id);

    // load the game using the returned info

  }

```
        catch (error){

            throw error;

        }

}
```

The await operator will ensure the next line of code is not executed until the login() function returns a user object. The getPlayerInfo() function is also preceded by the await operator. Once this function returns a result, it's assigned to the variable info, and this can then be used to load the actual game. A catch block is used to deal with any errors that may occur.

Generalized Functions

Callbacks can be used to build more generalized functions. One function can be written that accepts a callback instead of having many specific functions.

create a function that returns a random integer between two values that are provided as arguments, a and b, or if only 1 argument is provided, it will return a random integer between 1 and the argument provided:

Copy

```
function random(a,b=1) {

    // if only 1 argument is provided, we need to swap the values of a and b

    if (b === 1) {

        [a,b] = [b,a];

    }

    return Math.floor((b-a+1) * Math.random()) + a;

}


random(6);

<< 4


random(10,20);

<< 13
```

We can make the function more generic by adding a callback parameter:

```
function random(a,b,callback) {

    if (b === undefined) b = a, a = 1; // if only one argument is supplied, assume the lower limit is 1

        const result = Math.floor((b-a+1) * Math.random()) + a

    if(callback) {

        result = callback(result);

    }

    return result;

}
```

Functions that return Functions

```
function returnHello() {

    console.log('returnHello() called');

    return function() {

        console.log('Hello World!');

    }

}
```

To make use of the function that is returned, we need to assign it to a variable:

Copy

```
const hello = returnHello();
```

<< returnHello() called

Now we can invoke the 'Hello World' function by placing parentheses after the variable that it was assigned to:

Copy

```
hello()
```

<< Hello World!

Create a generic 'greeter' function that takes a particular greeting as a parameter, then returns a more specific greeting function:

Copy

```
function greeter(greeting = 'Hello') {

   return function() {

      console.log(greeting);

   }

}


const englishGreeter = greeter();

englishGreeter();

<< Hello


const frenchGreeter = greeter('Bonjour');

frenchGreeter();

<< Bonjour



const germanGreeter = greeter('Guten Tag');

germanGreeter();

<< Guten Tag
```

Closures

Function Scope

One of the key principles in creating closures is that an 'inner' function, which is declared inside another function, has full access to all of the variables declared inside the scope of the function in which it's declared (the 'outer' function). This can be seen in the example below:

```
function outer() {
```

```
    const outside = 'Outside!';

    function inner() {

        const inside = 'Inside!';

        console.log(outside);

        console.log(inside);

    }

    console.log(outside);

    inner();

}
```

The outer() function only has access to the variable outside, which was declared in its scope. The inner() function, however, has access to the variable inside, declared in its scope, but also the variable outside, declared outside its scope, but from within the outer() function.

Whenever a function is defined inside another function, the inner function will have access to any variables that are declared in the outer function's scope.

Returning Functions

A **closure** is formed when the inner function is returned by the outer function, maintaining access to any variables declared inside the enclosing function.

```
function outer() {

    const outside = 'Outside!';

    function inner() {

        const inside = 'Inside!';

        console.log(outside);

        console.log(inside);

    }

    return inner;

}
```

We can now assign a variable to the return value of the outer() function:

```
const closure = outer();
```

What makes this a closure is that it now has access to the variables created inside both the outer() and inner() functions, as we can see when we invoke it:

closure();

<< Outside!

Inside!

Generators

Special functions used to produce iterator that maintain the state of a value.

To define a generator function, an asterisk symbol (*) is placed after the function declaration:

```
function* exampleGenerator() {

// code for the generator goes here

}
```

This doesn't run any of the code, but returns a Generator object that can be used to create an iterator that implements a next() method that returns a value every time the next(0 method is called.

For example, we can create a generator to produce a Fibonacci-style number series (a sequence that starts with two numbers and the next number is obtained by adding the two previous numbers together), using the following code:

```
function* fibonacci(a,b) {

    let [ prev,current ] = [ a,b ];

    while(true) {

        [prev, current] = [current, prev + current];

        yield current;

    }

}
```

The code starts by initializing the first two values of the sequence, which are provided as arguments to the function. A while loop is then used, which will continue indefinitely due to the fact that it uses true as its condition, which will obviously always be true. Every time the iterator's next() method is called, the code inside the loop is run, and the next value is calculated by adding the previous two values together. The difference between the yield and the return keywords is that by using yield, the state of the value returned is remembered the next time yield is called. Hence, the current value in the Fibonacci sequence will be stored for use later. The execution of the loop is paused after every yield statement, until the next() method is called again.

Create a generator object:

const sequence = fibonacci(1,1);

The generator object is now stored in the sequence variable. It inherits a method called next(), which is then used to obtain the next value produced by the yield command:

Copy

sequence.next();

<< 2

sequence.next();

<< 3

sequence.next();

<< 5

It's also possible to iterate over the generator to invoke it multiple times:

Copy

```
for (n of sequence) {
   // stop the sequence after it reaches 100
   if (n > 10) break;
   console.log(n);
}
```

<< 8

<< 13

<< 21

<< 34

<< 55

<< 89

Functional Programming

Functional programming is a programming paradigm. Other examples of programming paradigms include object oriented programming and procedural programming. JavaScript is a multi-paradigm language, meaning that it can be used to program in a variety of paradigms and sometimes a combination of them.

Pure Functions

Pure functions follow these rules:

1. Return value should only depend on values provided as aruments and doesn't rely on values from somewhere else.
2. No side effects. It doesn't change any values or data elsewhere in the program.
3. Referential transparency- Same arguments will produce same result.

Must have at least one argument and a return value.

Higher Order functions

Functions that accept another function as an argument or return another function as a result, or both.

Closures are used a lot in higher-order functions because we can create a generic function that can be used to return more specific functions based on its arguments.

This is done by creating a closure around a function's arguments that keeps them 'alive' in a return function. For example, consider the following multiplier() function:

Copy

```
function multiplier(x){
   return function(y){
      return x*y;
   }
}
```

The multiplier() function returns another function that traps the argument x in a closure. This is then available to be used by the returned function.

We can now use this generic multiplier() function to create more specific functions, as can be seen in the example below:

Copy

doubler = multiplier(2);

This creates a new function called doubler(), which multiplies a parameter by the argument that was provided to the multiplier() function (which was 2 in this case). The end result is a doubler() function that multiplies its argument by two:

Copy

doubler(10);

<< 20

The multiplier() function is an example of a higher-order function. This means we can use it to build other, more specific functions by using different arguments. For example, an argument of 3 can be used to create a tripler() function that multiplies its arguments by 3:

Copy

tripler = multiplier(3);

tripler(10);

<< 30

Currying

A process that involves the partial application of functions. A function is curried when not all arguments have been supplied to the function so it returns another function that retains the arguments already provided and expects the remaing arguments that were omitted when the original function was called.

```
function multiplier(x,y) {

  if (y === undefined) {

    return function(z) {

    return x * z;

    }

  } else {

    return x * y;

  }
```

}

Now, if you found yourself frequently calculating the tax using the same rate of 22%, you could create a new curried function by providing just 0.22 as an argument:

calcTax = multiplier(0.22);

<< function (z){

   return x * z;

}

Ajax

Asynchronous, JavaScript, XML

A technique that allows web pages to communicate asynchronously with a server and dynamically updates web pages without reloading. Allows data to be sent and received in background as well as position of a page to be updated in response to user events while program continues to run. No longer static websites, but dynamic applications.

Clients and Servers

Client (web browser) requests a resource (usually a web page) from a server, which processes the request and sends back a response to the client.

Ajax allows JavaScript to request resources from a server on behalf of the client. Requested resources are usually JSON data or small text fragments or HTML rather than a whole web page.

Server is required when requesting resources using Ajax. Usually involves using a server-side language (PHP, Ruby, Node.js, .NET). To practice using Ajax, you can set up a local development server on your own computer, or request the files from an external website that uses cross-origin resource sharing (CORS).

Same-origin policy- Blocks all requests in browsers from a domain that is different from the page making the request. CORS use HTTP headers to indicate which domains can receive data.

Fetch API

Standard for requesting and sending data asynchronously across a network. It uses promises to avoid callback hell, and streamlines concepts that become cumbersome when using XMLHttpRequest object.

Basic Usage

Provides a global fetch() method with one mandatory argument: the URL of the resource you wish to fetch. Example:

fetch('https://example.com/data')

.then(// code that handles the response)

.catch(// code that runs if the server returns an error)

Response Interface

Deals with the object that is returned when the promise is fulfilled. Response objects have properties and methods that allow us to process the response.

ok property check to see if the response was successful. This is based on HTTP status code accessed using the **status** property. The code will usually be **200** if the response was successful, **201** if s resource was created, or **204** when the request is successful but no content is returned. The **ok** property will return true if the status property is between 200 and 299. Needs to be manually checked because promise will only be rejected in the case of a network error. Could be '404 page not found' which is still considered a successful request in terms of the promise.

We can use an if statement to check if the request was successful a throw an error if not:

const url = 'https:example.com/data';

fetch(url)

.then((response) => {

   if(response.ok) {

      return response;

   }

   throw Error(response.statusText);

})

.then(response => // do something with response)

.catch(error => console.log('There was an error!'))

Other properties of the Response Object:

headers
url
redirected
type

Can also contain methods that return promises that can be chained together.

Redirects

*currently no support in any browser

redirect() method used to redirect to another URL. Creates a new promise that resolves to the response from the redirected URL. Example:

fetch(url)

.then(response => response.redirect(newURL)); // redirects to another URL

.then(// do something else)

.catch(error => console.log('There was an error: ', error))

Text Responses

text() method takes a text from the reponsem reads it, then returns a promise that resolves to a USVSting object that can be treated as a string in JavaScript. Example:

fetch(url)

.then(response => response.text()); // transforms the text stream into a JavaScript string

.then(text => console.log(text))

.catch(error => console.log('There was an error: ', error))

File Responses

blob() method is used to read a file of raw data, such as an image or spreadsheet. Once read, it returns a promise that resolves with a **blob** object. Example:

fetch(url)

.then(response => response.blob()); // transforms the data into a blob object

.then(blob => console.log(blob.type))

.catch(error => console.log('There was an error: ', error))

JSON Responses

Most common for AJAX responses. **json()** method used to deal with these by transforming a stream of JSON data into a promise that resolves to a JavaScript object. Example:

fetch(url)

.then(response => response.json()); // transforms the JSON data into a JavaScript object

```
.then( data => console.log(Object.entries(data)) )

.catch( error => console.log('There was an error: ', error))
```

Object.entries() method can be used to view the key:value pairs in the returned object

Creating Response Objects

Mostly we will deal with a response object that is returned form a request we make. You can make your own response with a constructor function:

```
const response = new Response( 'Hello!', {

  ok: true,

  status: 200,

  statusText: 'OK',

  type: 'cors',

  url: '/api'

});
```

Request Interface

We can have more control over requests by providing a **Request** object as an argument.

Create using the **Request()** constructor and include these properties:

```
url
method
headers -Headers object
mode
chache
credentials
redirect
```

The Web is built upon the Hypertext Transfer Protocol, or HTTP. When a client (usually a browser) makes a request to a server, it contains information about which HTTP verb to use. HTTP verbs, also known as HTTP methods are the what HTTP uses to tell the server what type of request is being made, which then determines the server will deal with the request.

The five most commonly used verbs when dealing with resources on the web are:

GET requests to retrieve resources

POST requests, usually used to create a resource but can actually perform any task

PUT requests to upsert, which means insert a resource or update it entirely

PATCH requests to make partial updates to a resource

DELETE requests to delete a resources.

By default, a link in a web page will make a GET request. Forms are also submitted using a GET request by default, but they will often use a POST request.

A constructor function is used to create a new Request object. An example is shown below:

```
const request = new Request('https://example.com/data', {
    method: 'GET',
    mode: 'cors',
    redirect: 'follow',
    cache: 'no-cache'
});
```

Once the Request object is assigned to a variable, it can then be used as the parameter of the fetch() method:

```
fetch(request)
.then( // do something with the response )
.catch( // handle any errors)
```

Alternatively, you can enter the URL and object directly as arguments of the fetch() method, without having to create a Request object:

```
fetch('https://example.com/data', {

    method: 'GET',

    mode: 'cors',

    redirect: 'follow',

    cache: 'no-cache'

})

.then( // do something with the response )

.catch( // handle any errors)
```

Headers Interface

HTTP header are used to pass on any additional info about a request or response. Info includes file-type of the resource, cookie info, authentication info, when the resource was modified. **Headers interface** creates Headers object, then added as a property of Request and Response objects.

New instance of Headers:

```
const headers= new Headers();
```

Constructor function with an option argument containing initial header values:

```
const headers = new Headers({ 'Content-Type': 'text/plain', 'Accept-Charset' : 'utf-8', 'Accept-Encoding':'gzip,deflate' })
```

Headers object includes the following properties and methods:

```
has()
get()
set()
append()
delete()
keys(), values(), and entries()
```

Putting it all together

We can use the Headers, Request and Response objects to put together a typical example that sets up the URL, Request and Headers before calling the fetch() method:

```
const url = 'https:example.com/data';

const headers = new Headers({ 'Content-Type': 'text/plain', 'Accept-Charset' : 'utf-8', 'Accept-Encoding':'gzip,deflate' })


const request = (url,{

  headers: headers

})


fetch(request)

.then( function(response) {

  if(response.ok) {

    return response;

  }

  throw Error(response.statusText);

})

.then( response => // do something with response )

.catch( error => console.log('There was an error!') )
```

FormData

Fetch API includes the FormData interface, makes it easier to submit info in forms using Ajax.

FormData instance with constructor function:

```
const data = new FormData();
```

FormData interface helps reduce amount of code needed:

```
const form = document.forms['todo'];


form.addEventListener('submit', addTask, false);


function addTask(event) {

  event.preventDefault();
```

```
const task = new FormData(form);

const url = `http://echo.jsontest.com/id/1/title/${form.task.value}`;

const headers = new Headers({

   'Accept': 'application/json',

   'Content-Type': 'application/json'

});

const request = new Request(url,

{

   method: 'POST',

   mode: 'cors',

   header: headers,

   body: JSON.stringify(task)

}

)


fetch(request)

.then( response => response.json() )

.then( data => console.log(`${data.title} saved with an id of ${data.id}`) )

.catch( error => console.log('There was an error:', error))


}
```

The FormData interface really comes into its own when a form contains files to upload. This was a notoriously difficult task in the past, often requiring the use of Flash or another third-party browser plugin to handle the upload process. The FormData instance will automatically create the necessary settings required, and take care of all the hard work if any file uploads are present in the form.

You can find more information about the FormData interface in this SitePoint article by Craig Buckler and on the Mozilla Developer Network

A Living Standard

Fetch API is experimental and subject to change. Living standards often stay relatively stable. Instead of Fetch API, you can use a library like jQuery to take care of Ajax requests.