JavaScript

Mocha? LiveScript? ECMAScript? JavaScript?

Forget it. Call it what you want.

JavaScript - History

- The language formerly known as LiveScript, standardized by ECMA, and commonly referred to as JavaScript began life under the development name Mocha.
- ECMAScript is a language standardized by ECMA (European Computer Manufacturers Association) [Since 1994 "Ecma" and no longer an acronym].
- JavaScript supports imperative/procedural, functional, and prototype-based object-oriented coding paradigms.

JavaScript - History

- It was developed by and for Netscape, to be part of its Netscape Navigator web browser (may it rest in peace). At the last minute the name was changed to "JavaScript" as a marketing ploy; Java was 'in' during that time and they wanted to take advantage of the popularity.
- It has NOTHING to do with Java. It is about as different from Java as a language can be.

JavaScript - History

- The first version was written in 10 days, by Brendan Eich in 1995.
- Many programmers believe that JavaScript has only recently began to be a server-side technology as well as a browser technology, but this is not strictly true. Netscape created a server-side version shortly after releasing the original browser based version.
- Microsoft has also had server-side with JScript for many years.

JavaScript - Basics

- It is a dynamic, weakly typed, prototype-based language.
- The thing that trows many people off is the **Prototype-based** part.
- We will discuss this in a few minutes.

JavaScript - Variables

We create a variable in JS with either the `const`, `let`, or `var` keywords.

- `const` should always be used for a variable with an unchanging value.
- `let` should be used for variables otherwise. Creates a locally (block) scoped variable.
- `var` for the most part should **not** be used as it does not limit scope to blocks (uses function scope).

JavaScript - Variables

For instance (with `let`):

```
> for(let i=0; i<5; ++i){
... console.log(i)
... }
0
1
2
3
4
undefined
> console.log(i)
ReferenceError: i is not defined
>
```

JavaScript - Variables

With `var`:

```
> for(var i=0; i<5; ++i){
... console.log(i)
... }
0
1
2
3
4
undefined
> console.log(i)
5
```

JavaScript - Variables

And `const`:

```
> for(const i=0; i<5; ++i){
... console.log(i);
... }
0
```

JavaScript - Objects

In JavaScript objects are hash maps (key-value pairs). The keys are strings and the values are objects (functions are objects too).

Since JavaScript is dynamic we can add to the hash map at any time:

JavaScript - Object Properties

- A property of an object is a variable or a function.
- A property that has not been created is `undefined`, not `null`. We can set a property equal to `null` to indicate it has no value, but if we are checking to see if a property exists we would use `undefined`:

```
> var one = { name: 'CIS 343' };
undefined
> one.description
undefined
> one.description === undefined
true
```

JavaScript - Objects

```
var myComputer = {};
myComputer.ram = '16G'
myComputer.cores = 4
```

Or

```
var myComputer = {};
myComputer['ram'] = '16G'
myComputer['cores'] = 4
```

JavaScript - Objects

Creating objects in JavaScript can happen in multiple ways.

We can use an **Object Literal**:

```
let student = {
        name: "Josephine",
        g: 123456,
        gpa: 4.0
}
```

We usually won't use this if we need to create multiple objects that are similar.

JavaScript - Objects

We can use **Factory Functions**:

```
let Student = function(_name, _g, _gpa){
        return {
                name: _name,
                g: _g,
                gpa: _gpa
        }
}
let joey = Student("Josephine", 123456, 4.0);
```

JavaScript - Objects

We can use the **new** operator with a **Constructor Function**:

```
function Student(name, g, gpa){
        this.name = name;
        this.g = g;
        this.gpa = gpa;
}

let joey = new Student("Josephine", 123456, 4.0);
```

Notice here that we are using the **new** keyword. Since we are using the `new` keyword we don't need to return the object; it will automatically happen.

JavaScript - Objects

We can use the Object.create() method. With this we can make the new object a prototype of another:

```
let Student = {
        name: '',
        g: 0,
        gpa: 4.0
}
let joey = Object.create(Student);
```

This works well when we want to create multiple from a single prototype.

JavaScript - Objects

As of the ES6 Standard we have the **class** keyword:

```
class Student {
        constructor(name, g, gpa){
                this.name = name;
                this.g = g;
                this.gpa = gpa;
        }
}

let joey = new Student("Josephine", 123456, 4.0);
```

JavaScript - Objects

Note!

- This doesn't change anything; it is just syntactic sugar. We are merely creating an object in a way that looks like the classical way. Also, this does not mean that `Student` is the prototype of `joey`. We have simply created multiple Student objects by calling the function.
- I will likely use each of these versions in class; if I know that I'm not going to need multiple similar objects I usually don't worry with the overhead of simulating the classical method.

JavaScript - Prototypes

Let's remember our definitions here:

- **Class** - a blueprint for an object. Not instantiated. JavaScript does **not** have these.
- **Object** - an instance of a class. Instantiated (hence it is an "instance").

JavaScript - Prototypes

- Typicaly object-oriented programming provides code re-use via an object inheriting from a parent *class*.
- The child also holds the parent data (though you may still not be able to access it).

JavaScript - Prototypes

- In JavaScript there is no "class" (yes, there is syntactic sugar that makes it seem JavaScript has classes, but it really doesn't). There are only objects.
- One object may be a prototype of another object.

JavaScript - Prototypes

Consider:

```
> var person = { name: 'ira', height: '6 ft 0 in' }
undefined
> var occupation = { title: 'Professor', skill: 'Wizardry' }
undefined
> Object.setPrototypeOf(person, occupation)
{ name: 'ira', height: '6 ft 0 in' }
> person.title
'Professor'
>
```

JavaScript - Prototypes

- Here we created a new object called person, and set the prototype of that person object (not class) to occupation. An object can be a prototype of only one object at a time, but by being a prototype it gains the properties of the prototype.
- However, prototypes can be chained:

JavaScript - Prototypes

```
> var ira = {}
undefined
> Object.setPrototypeOf(ira, person)
{}
> ira.name
'ira'
> ira.title
'Professor'
>
```

- Here, the object `ira` was set to the prototype of `person`. As it is a prototype of `occupation`, `ira` has the properties of all the prior objects in the chain.

JavaScript - Prototypes

- It is important to understand that an object does not hold a property it gets from a prototype. Note the following:

```
> var first = { name: 'first' }
undefined
> first
{ name: 'first' }
> var second = { type: 'second' }
undefined
> second
{ type: 'second' }
> Object.setPrototypeOf(second, first)
{ type: 'second' }
> second
{ type: 'second' }
> second.name
'first'
>
```

JavaScript - Prototypes

- This isn't like classical inheritance; `second` doesn't now have an instance variable 'name'. It simple inherits the properties of `first`.
- If we want `second` to have 'name' set to 'two' we can do that; but that is going to override the property we got from the prototype.

JavaScript - Prototypes

```
> second.name
'first'
> second.name='two'
'two'
> second
{ type: 'second', name: 'two' }
>
```

Notice that `second` now has a `name` property? We have overriden (basically hidden) the prototype property because our object now holds its own with the same name.

JavaScript - Prototypes

If we ever wish to add to the prototype, all objects that use the prototype gain access to the new properties as well:

```
> var Computer = { ram:'16G', cores:4}
undefined
> var Laptop = { portable:'yes' }
undefined
> Object.setPrototypeOf(Laptop, Computer)
{ portable: 'yes' }
> Laptop.cpu_speed
undefined
> Computer.cpu_speed = '3.7G'
'3.7G'
> Laptop.cpu_speed
'3.7G'
>
```

JavaScript - Functions

Functions, just like any other object, can be a property of an object. We define functions with the `function` keyword:

```
myComputer.startup = function(message){ console.log(message); }
[Function]
myComputer.startup("Booting now!")
Booting now!
```

JavaScript - Functions

We can create functions in a lot of ways in JavaScript:

```
// "Normal" syntax
function square(x){
        return x * x;
}
```

```
// Function expression syntax
const square = function(x){
        return x * x;
}
```

JavaScript - Arrow Functions

- Creating a function with the following (called arrow syntax) is allowed as well:

```
const square = (x) => { return x * x };
```

- There are some caveats to arrow syntax though:
- Arrow functions don't bind to `this` or `super`.
- You can't create constructor functions with them.

(this is not an exhaustive list of caveats, but covers the level we are currently in class).

JavaScript - Closures

It is often useful to have a function within a function. This is called a **closure**.

- Inner functions have access to outer function symbols.
- Outer function do **not** have access to inner function symbols.
- Allows us to simulate private variables (encapsulation!)

Consider:

```
const createAccount = function(customer){
    let money = 0;

    return {
        getMoney: function(){
                return money;
        },
        addMoney: function(amount){
                if(amount > 0){
                        money += amount;
                }
        },
        getCustomer(){
                return customer;
        }
    }
}
```

JavaScript - Asynchronous Programming

The biggest issue students often have with JavaScript programming is that it is **asynchronous**.

- We are used to a model where a line of code doesn't run until the previous line has completed.
- This is not always the case with JavaScript!

JavaScript - Asynchronous Programming

JavaScript was created with a focus on the Web.

- inspect the web page for https://www.espn.com (or another large page)
- many, many objects make up the page
- the objects come from a lot of different places
- we have no way of knowing how long we might wait for each resources

If we handled this synchronously, page load would be horrible!

JavaScript - Asynchronous Programming

JS (both in browsers and natively such as NodeJs) has libraries that perform functions for us that might be "slow".

- "slow" also means indeterminate time
- fetching a remote resource might be fast, but there is no way to tell beforehand
- anything that JS asks one of those libraries to do for us will be asynchronous
- these libraries are provided by **WebAPI**s (browser) or C++ libraries (node).

JavaScript - Asynchronous Programming

This means that code like the following won't behave as you might suspect:

```
function funkyOne(){
        console.log( "Yo.  VIP.  Kick it." );
}

function getData(){
        setTimeout(funkyOne, 3000);
        return 42;
}

const value = getData();
console.log(value);
console.log("Jiggy.")
```

JavaScript - Asynchronous Programming

It turns out the `setTimeout` is a function that is provided by one of those libraries. The output works if you understand a few concepts:

- **Call Stack** - the stack that our functions are place on as they are called. This is part of JS.
- **Event Loop** - Not part of JS. The part of the runtime (either the browser or node) that continuously processes functions on the stack, *and then* checks the message queue.
- **Message Queue** - An area for messages from libraries to wait until the event loop can check them. Not part of JS.

JavaScript - Asynchronous Programming

So why did we get the output we got from before?

- getData() was called and placed on the call stack
- getData() called setTimeout(funkyOne, 3000)
- getData() returned 42
- 42 was printed
- Jiggy was printed
- After some time the library handling the setTimeout function called funkyOne()
- funkyOne was placed on the stack
- funkyOne was processed and the message printed

JavaScript - Asynchronous Programming

Sometimes people say "oh, I see. JavaScript is multithreaded."

No.

It could be. Often it is not.

JS and the WebAPIs or C++ libraries are different things and should be treated as such.

JavaScript - Asynchronous Programming

This can make programming tricky.

- How do we know to run some new code once the old code has finished?

Since JS allows first-class-functions we could use a callback.

```
function runMe(cb){
        setTimeout( function(cb) {
                console.log("Two");
                cb();
        }, 1000);
}
three = function(){
        console.log("Three");
}
runMe(three);
console.log("One")
```

JavaScript - Promises

Too many callbacks can make a mess and can lead to what many refer to as "callback he||".

Intead, we can use **promises**.

A promise is an object that we can return that says, "I'll give the answer later.".

JavaScript - Promises

We can write a promise like this:

```
function slowCode(){}
        return new Promise((resolve, reject) => {
                ... DO SOMETHING THAT TAKES A WHILE ...
                if(SUCCESS){
                        resolve()
                } else {
                        reject()
                }
        });
}
```

JavaScript - Promises

A promise takes one or more functions. Above we used one that takes (resolve, reject) as parameters.

- `resolve` is a function that takes a single argument. It is called when the promise is fulfilled.
- `reject` takes a single argument. It is called if the promise fails.

JavaScript - Promises

Here's a common problem with asynchronous programming. We need to pass the information returned from one function to another.

```
... Code runs ...
fetch('https://v2.jokeapi.dev/joke/Any\?blacklistFlags\=nsfw,religious,racist,sexist,explicit')
... Code runs ... Fetch is being handled by a library
```

- The line after the fetch will run immediately.
- How do we get the result?

JavaScript - Promises

This won't work...

```
import fetch from 'node-fetch';

fetch('https://v2.jokeapi.dev/joke/Any\?blacklistFlags\=nsfw,religious,racist,sexist,explicit')
```

- Nothing will be printed

JavaScript - Promises

How about this?

```
import fetch from 'node-fetch';

let result = fetch('https://v2.jokeapi.dev/joke/Any\?blacklistFlags\=nsfw,religious,racist,sexist,explicit')
console.log(result)
```

No... the output gives

```
Promise { <pending> }
```

Why?

JavaScript - Promises

A promise is a way for code to say it doesn't have a result now but will give you one when it can.

A promise has state:

- fulfilled - it did its job and gave you the result
- pending - neither fulfilled or rejected yet
- rejected - code couldn't complete and failed

JavaScript - Promises

So here we had a promise that was pending. How do we know when it is fulfilled (and thus that our data is available?)?

We can use a `.then()`.

```
let url = 'https://v2.jokeapi.dev/joke/Any\?blacklistFlags\=nsfw,religious,racist,sexist,explicit'
fetch(url).then( ... DO SOMETHING ...)
```

JavaScript - Promises

Inside of the `.then()` we put a function and give it the data received from the async function.

- There is also a `.catch()` for catching errors and a `.finally()` for code running after either of the others.

```
let url = 'https://v2.jokeapi.dev/joke/Any\?blacklistFlags\=nsfw,religious,racist,sexist,explicit'

fetch(url)
        .then( ... DO SOMETHING ...)
        .catch( ... HANDLE ERROR ...)
        .finally( .. DO SOMETHING ELSE ...)
```

JavaScript - Promises

Sometimes we need to chain promises. For instance, the fetch code above eventually returns a `Response` object (if the promise succeeds).

- We would pass that data to a `.then()` function:

```
fetch(url).then(r => console.log(r))
```

Output:

```
Response {
  size: 0,
  [Symbol(Body internals)]: {
    body: PassThrough {
      _readableState: [ReadableState],
      _events: [Object: null prototype],
      _eventsCount: 5,
      _maxListeners: undefined,

... And so on...
```

This is not what we want...

JavaScript - Promises

We need to pull the part of the response we want out. I happen to know that this response has a JSON payload so I'm going to ask for that.

- That is something a library does for us.
- Which means processing the response will give us a promise...

```
fetch(url).then(response => response.json())
          .then(data => console.log(data))
```

```
{
  error: false,
  category: 'Programming',
  type: 'twopart',
  setup: 'why do python programmers wear glasses?',
  delivery: "Because they can't C.",
  flags: {
    nsfw: false,
    religious: false,
    political: false,
    racist: false,
    sexist: false,
    explicit: false
  },
  safe: true,
  id: 294,
  lang: 'en'
}
```

JavaScript Promises

Chaining `.then()`s together can get ugly. These days we have the `async` and `await` keywords that allow us to make code that looks more like synchronous code:

```
async function getJoke(){
        let response = await fetch('https://v2.jokeapi.dev/joke/Any\?blacklistFlags\=nsfw,religious,racist,sexist,explicit');
        let json = await response.json();
        console.log(json);
}
```

- This looks more like the code we are used to.
- Just syntactic sugar.

JavaScript Promises

The best part of this syntax is we can surround this code with try-catch blocks:

```
#!/usr/bin/env node

import fetch from 'node-fetch';

try{
        let response = await fetch('https://v2.jokeapi.dev/joke/Any\?blacklistFlags\=nsfw,religious,racist,sexist,explicit')
        let json = await response.json()
        console.log(json)
} catch(err){
        console.log("Uh-oh...");
}
```

JavaScript

How do I know when a function will return a promise?

> How do we always know what a function will return?

JavaScript

We read the docs!

For instance, look at the documentation for `fetch()`:

https://developer.mozilla.org/en-US/docs/Web/API/fetch

JavaScript

Let's practice!