Notes on promises

https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html

Basic Promise Usage https://davidwalsh.name/promises

The new Promise() constructor should only be used for legacy async tasks, like usage of setTimeout or XMLHttpRequest. A new Promise is created with the new keyword and the promise provides resolve and reject functions to the provided callback:

```
var p = new Promise(function(resolve, reject) {
        // Do an async task async task and then...
        if(/* good condition */) {
                resolve('Success!');
        }
        else {
                reject('Failure!');
        } });
  p.then(function() {
        /* do something with the result */
 }).catch(function() {
        /* error :( */
 })
```

It's up to the developer to manually call resolve or reject within the body of the callback based on the result of their given task.
Since a promise is always returned, you can always use the then and catchmethods on its return value!

then

All promise instances get a then method which allows you to react to the promise. The first then method callback receives the result given to it by the resolve() call:
The then callback is triggered when the promise is resolved. You can also chain then method callbacks:

```
new Promise(function(resolve, reject) {
        // A mock async action using setTimeout
        setTimeout(function() { resolve(10); }, 3000);
 })
.then(function(num) { console.log('first then: ', num); return num * 2; })
.then(function(num) { console.log('second then: ', num); return num * 2; })
.then(function(num) { console.log('last then: ', num);});
  // From the console:
 // first then:  10
 // second then:  20
 // last then:  40
```

Each then receives the result of the previous then's return value.
If a promise has already resolved but then is called again, the callback immediately fires. If the promise is rejected and you call then after rejection, the callback is never called.

catch

The catch callback is executed when the promise is rejected:

```
new Promise(function(resolve, reject) {
        // A mock async action using setTimeout
        setTimeout(function() { reject('Done!'); }, 3000);
 })
 .then(function(e) { console.log('done', e); })
 .catch(function(e) { console.log('catch: ', e); });
  // From the console:
 // 'catch: Done!'
```

What you provide to the reject method is up to you. A frequent pattern is sending an Error to the catch:

```
reject(Error('Data could not be found'));
```

Promise.all

Think about JavaScript loaders: there are times when you trigger multiple async interactions but only want to respond when all of them are completed -- that's where Promise.all comes in. The Promise.all method takes an array of promises and fires one callback once they are all resolved:

```
Promise.all([promise1, promise2]).then(function(results) {
        // Both promises resolved
 })
.catch(function(error) {
        // One or more promises was rejected
 });
```

We have a problem with promises:

Even if you never expect an error, it's always prudent to add a catch(). It'll make your life easier, if your assumptions ever turn out to be wrong.

As I said before, the magic of promises is that they give us back our precious return and throw. But what does this actually look like in practice?

Every promise gives you a then() method (or catch(), which is just sugar for then(null, ...)). Here we are inside of a then() function:

```
somePromise().then(function () {

  // I'm inside a then() function!

});
```

What can we do here? There are three things:

1. return another promise
2. return a synchronous value (or undefined)
3. throw a synchronous error

That's it. Once you understand this trick, you understand promises.

1. Return another promise

This is a common pattern you see in the promise literature, as in the "composing promises" example above:

```
getUserByName('nolan').then(function (user) {

  return getUserAccountById(user.id);

}).then(function (userAccount) {

  // I got a user account!

});
```

Notice that I'm returning the second promise – that return is crucial. If I didn't say return, then the getUserAccountById()would actually be a *side effect*, and the next function would receive undefined instead of the userAccount.

2. Return a synchronous value (or undefined)

Returning `undefined` is often a mistake, but returning a synchronous value is actually an awesome way to convert synchronous code into promisey code. For instance, let's say we have an in-memory cache of users. We can do:

```
getUserByName('nolan').then(function (user) {
  if (inMemoryCache[user.id]) {
    return inMemoryCache[user.id];    // returning a synchronous value!
  }
  return getUserAccountById(user.id); // returning a promise!
}).then(function (userAccount) {
  // I got a user account!
});
```

Isn't that awesome? The second function doesn't care whether the `userAccount` was fetched synchronously or asynchronously, and the first function is free to return either a synchronous or asynchronous value.

Unfortunately, there's the inconvenient fact that non-returning functions in JavaScript technically return `undefined`, which means it's easy to accidentally introduce side effects when you meant to return something.

For this reason, I make it a personal habit to *always return or throw* from inside a `then()` function. I'd recommend you do the same.

3. Throw a synchronous error

Speaking of `throw`, this is where promises can get even more awesome. Let's say we want to `throw` a synchronous error in case the user is logged out. It's quite easy:

```
getUserByName('nolan').then(function (user) {
  if (user.isLoggedOut()) {
    throw new Error('user logged out!'); // throwing a synchronous error!
  }
  if (inMemoryCache[user.id]) {
    return inMemoryCache[user.id];       // returning a synchronous value!
  }
  return getUserAccountById(user.id);    // returning a promise!
}).then(function (userAccount) {
  // I got a user account!
}).catch(function (err) {
  // Boo, I got an error!
});
```

Our `catch()` will receive a synchronous error if the user is logged out, and it will receive an asynchronous error if *any of the promises are rejected*. Again, the function doesn't care whether the error it gets is synchronous or asynchronous.

This is especially useful because it can help identify coding errors during development. For instance, if at any point inside of a `then()` function, we do a `JSON.parse()`, it might throw a synchronous error if the JSON is invalid. With callbacks, that error would get swallowed, but with promises, we can simply handle it inside our `catch()` function.