Promises, Iterators, and Generators

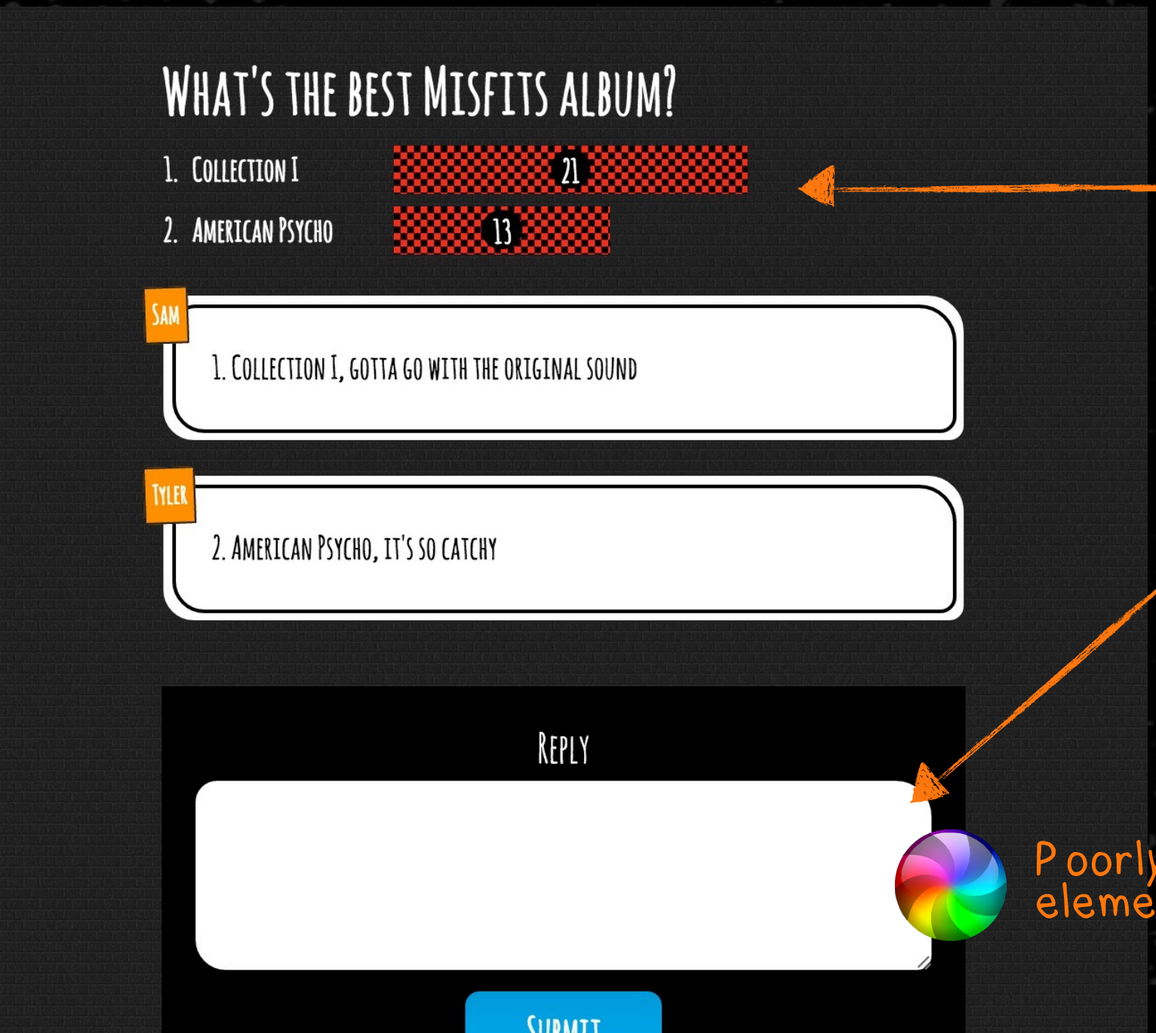
Level 6

Promises

Level 6 – Section 1

Fetching Poll Results From the Server

It's very important to understand how to work with JavaScript's **single-thread model**. Otherwise, we might accidentally **freeze** the entire app, to the detriment of user experience.



A script requests poll results from the server...

...and while the script waits for the response, users must be able to interact with the page.

Poorly written code can make other elements on the page unresponsive

Avoiding Code That Blocks

Once the browser **blocks** executing a script, it stops running other scripts, rendering elements, and responding to user events like keyboard and mouse interactions.

Synchronous style functions wait for return values

```
let results = getPollResultsFromServer("Sass vs. LESS");
ui.renderSidebar(results);
```

Page freezes until a value is returned from this function

In order to **avoid blocking** the main thread of execution, we write non-blocking code like this:

Asynchronous style functions pass callbacks

```
getPollResultsFromServer("Sass vs. Less", function(results){
  ui.renderSidebar(results);
});
```

Passing Callbacks to Continue Execution

In c**ontinuation-passing style** (CPS) async programming, we tell a function how to continue execution by passing callbacks. It can grow to **complicated nested code**.

```
getPollResultsFromServer(pollName, function(error, results){
  if(error){ //.. handle error }
  //...
  ui.renderSidebar(results, function(error){
    if(error){ //.. handle error }
    //...
    sendNotificationToServer(pollName, results, function(error, response){
      if(error){ //.. handle error }
      //...
      doSomethingElseNonBlocking(response, function(error){
        if(error){ //.. handle error }
        //...
      });
  });
});
```

When nested callbacks start to grow, our code becomes harder to understand

The Best of Both Worlds With Promises

ES2015

A Promise is a new abstraction that allows us to write async code in an easier way.

```
getPollResultsFromServer("Sass vs. LESS")
  .then(ui.renderSidebar)
  .then(sendNotificationToServer)
  .then(doSomethingElseNonBlocking)
  .catch(function(error){
      console.log("Error: ", error);
  });
```

Still non-blocking, but not using nested callbacks anymore

Let's learn how to create ***Promises!***

Creating a New Promise Object

ES2015

The Promise constructor function takes an anonymous function with 2 callback arguments known as **handlers**.

```
function getPollResultsFromServer(pollName){

  return new Promise(function(resolve, reject){
    //...
    resolve(someValue);

    //...
    reject(someValue);
  });
};
```

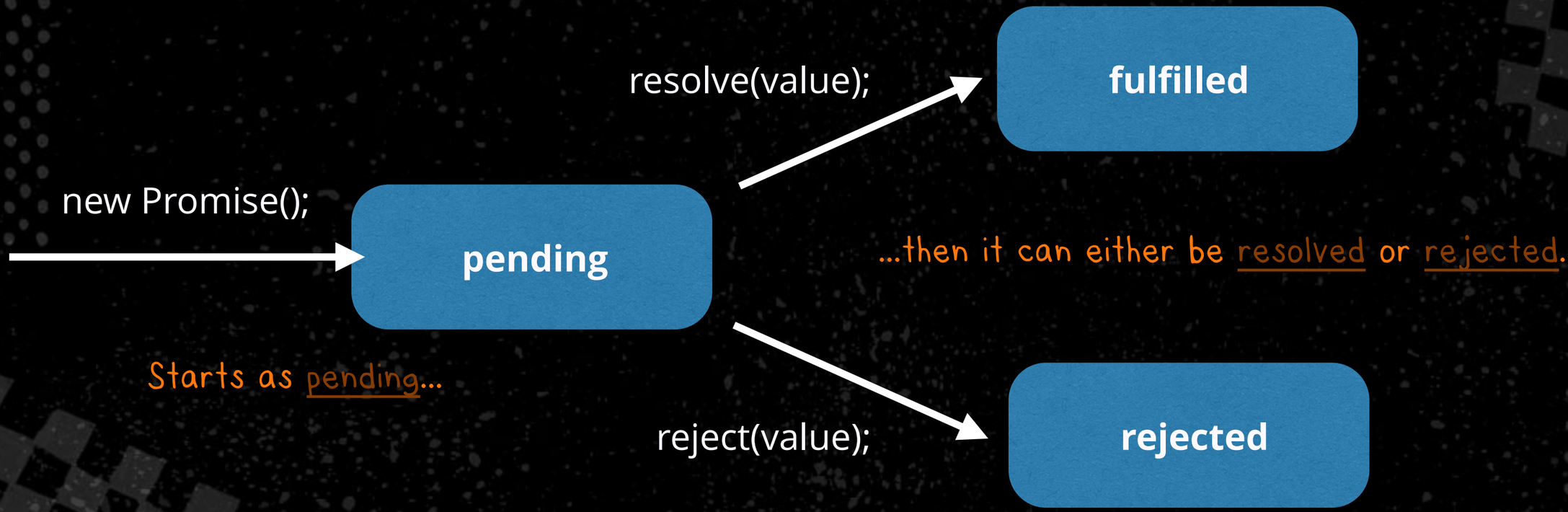
Handlers are responsible for either resolving or rejecting the Promise

Called when the non-blocking code is done executing

Called when an error occurs

The Lifecycle of a Promise Object

Creating a new Promise automatically sets it to the **pending** state. Then, it can do 1 of 2 things: become **fulfilled** or **rejected**.



Returning a New Promise Object

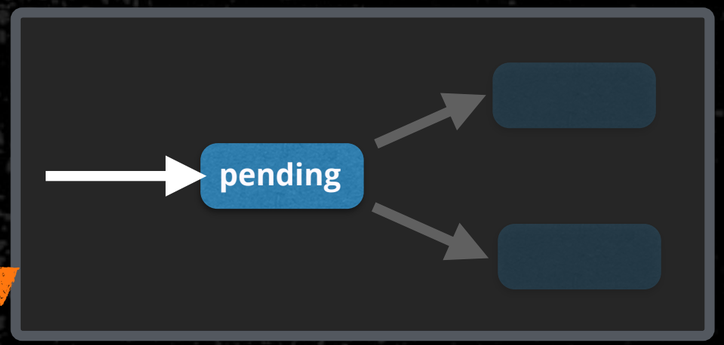
A Promise represents a **future value**, such as the eventual result of an **asynchronous** operation.

```
let fetchingResults = getPollResultsFromServer("Sass vs. Less");
```

Not the actual result, but a Promise object

No longer need to pass a callback function as argument

Starts on pending state

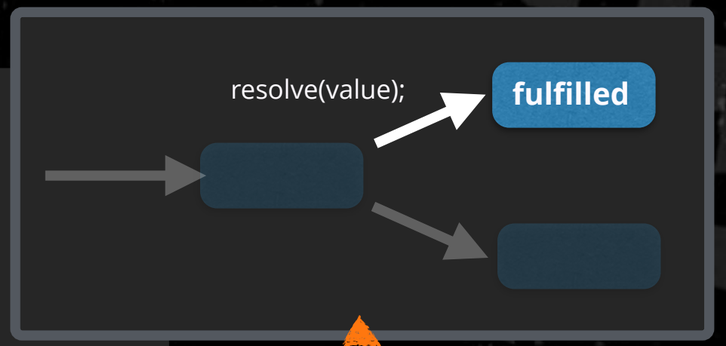


Resolving a Promise

Let's wrap the *XMLHttpRequest* object API within a Promise. Calling the *resolve()* handler moves the Promise to a **fulfilled** state.

```
function getPollResultsFromServer(pollName){

  return new Promise(function(resolve, reject){
    let url = `/results/${pollName}`;
    let request = new XMLHttpRequest();
    request.open('GET', url, true);
    request.onload = function() {
      if (request.status >= 200 && request.status < 400) {
        resolve(JSON.parse(request.response));
      }
    };
    //...
    request.send();
  });
};
```



Resolving a Promise moves it to a fulfilled state

We call the resolve() handler upon a successful response

Reading Results From a Promise

We can use the *then()* method to read results from the Promise once it's resolved. This method takes a function that will only be invoked once the Promise is **resolved**.

```
let fetchingResults = getPollResultsFromServer("Sass vs. Less");
fetchingResults.then(function(results){
  ui.renderSidebar(results);
});
```

This function renders HTML to the page

This is the argument previously passed to resolve()

```
function getPollResultsFromServer(pollName){
    //...
        resolve(JSON.parse(request.response));
    //...
};
```

Removing Temporary Variables

We are currently using a **temporary variable** to store our Promise object, but it's not really necessary. Let's replace it with **chaining function calls**.

```
let fetchingResults = getPollResultsFromServer("Sass vs. Less");
fetchingResults.then(function(results){
  ui.renderSidebar(results);
});
```

Temporary variable is unnecessary

Same as this

```
getPollResultsFromServer("Sass vs. Less")
  .then(function(results){
    ui.renderSidebar(results);
  });
```

Chaining Multiple Thens

We can also chain multiple calls to *then()* — the **return value** from 1 call is passed as argument to the next.

```
getPollResultsFromServer("Sass vs. Less")
  .then(function(results){
    return results.filter((result) => result.city === "Orlando");
  })
  .then(function(resultsFromOrlando){
    ui.renderSidebar(resultsFromOrlando);
  });
```

Only returns poll results from Orlando

The return value from one call to then...

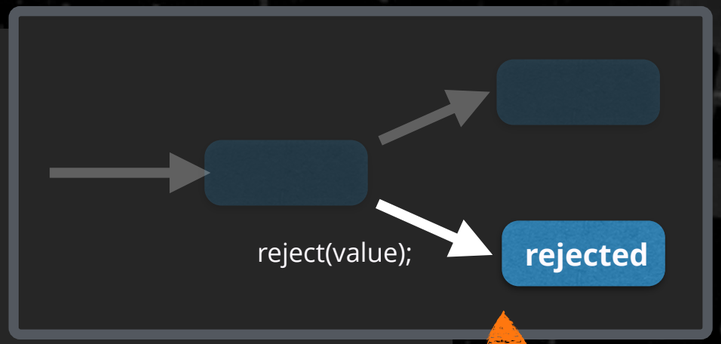
...becomes the argument to the following call to then.

Rejecting a Promise

We'll call the *reject()* handler for **unsuccessful status codes** and also when the *onerror* event is triggered on our request object. Both move the Promise to **a rejected state**.

```
function getPollResultsFromServer(pollName){

  return new Promise(function(resolve, reject){
   //...
   request.onload = function() {
      if (request.status >= 200 && request.status < 400) {
        resolve(request.response);
      } else {
        reject(new Error(request.status));
      }
    };
    request.onerror = function() {
      reject(new Error("Error Fetching Results"));
    };
    //...
```

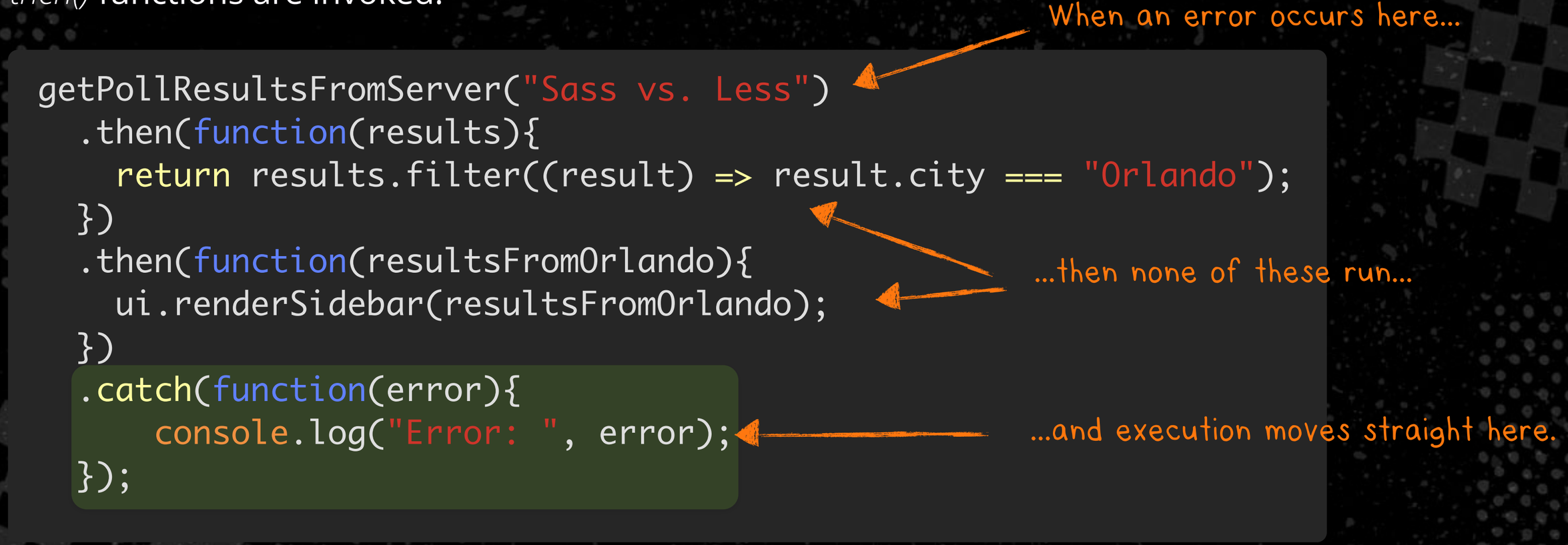


Rejecting a Promise moves it to a rejected state

We call the reject() handler, passing it a new Error object

Catching Rejected Promises

Once an error occurs, execution moves immediately to the *catch()* function. None of the remaining *then()* functions are invoked.




```
getPollResultsFromServer("Sass vs. Less")
  .then(function(results){
    return results.filter((result) => result.city === "Orlando");
  })
  .then(function(resultsFromOrlando){
    ui.renderSidebar(resultsFromOrlando);
  })
  .catch(function(error){
      console.log("Error: ", error);
  });
```

Passing Functions as Arguments

We can make our code more succinct by passing function arguments to *then*, instead of using anonymous functions.

```
function filterResults(results){ //... }

let ui = {
  renderSidebar(filteredResults){ //... }
};

getPollResultsFromServer("Sass vs. Less")
  .then(filterResults)
  .then(ui.renderSidebar)
  .catch(function(error){
      console.log("Error: ", error);
  });
```

Remember the new method initializer shorthand syntax?

Passing function arguments make this code easier to read

Still catches all errors from previous calls