ASYNC

JS code runs line by line

DOM events, trigger outside this

- And are handled once code execution is done

That triggering is **asynchronous** (async)

- You won't know when it happens
- It **won't** happen in the middle of code execution

Same is true for service calls

- start now,
  done...sometime

CALLBACKS

Async events are handled with callbacks

For DOM events you register a handler/listener

- It is called when appropriate

For Service calls, same idea

Also for filesystem calls on Node

Or Database interactions on Node

PYRAMID OF DOOM

When you have nested async calls:

- Do A
  - When A is done, do B
    - When B is done, do C
      - etc

It gets ugly, fast

Known as the **Pyramid of Doom** because the nested callbacks make an indented triangle

PROMISES

Promises are a way to track callbacks

An object that you can pass callbacks to

Gives you a NEW promise object when you do

You can chain them

Callbacks can be added even to completed promise

Still callbacks

SIMPLE PROMISE EXAMPLE

```
console.log(1);
returnsAPromise().then( () => console.log(2) );
console.log(3);
```

always logs `1 3 2`. **Always**

Why?

CHAINED EXAMPLE

```
returnsAPromise()
  .then( () => console.log(1) )
  .then( () => console.log(2) );
```

Always `1 2`. **Always**. Why?

RESOLVE VALUES

Promises might "resolve" with a value

- This value is passed to any callbacks
- This is **NOT** returned by the `then()` call

```
const promise = Promise.resolve("hi");
const value1 = promise.then(
  (text) => console.log(`callback: ${text}`)
);
console.log(`from then: ${value1}`);
```

```
from then: [object Promise]
callback: hi
```

Remember: `then()` returns a new promise

Golden rule: To use a value from async, you must stay async

CHAINING RETURNS

When a callback returns a value

- Becomes the resolve value of promise of that `then()`

```
const result = Promise.resolve(1)
  .then( val => {
    console.log(val);
    return val+1;
  })
  .then( val => {
    console.log(val);
    return val+1;
  })
  .then( val => {
    console.log(val);
    return val+1;
  });
```

What is `result`?

CHAINING

```
const one = Promise.resolve();
const two = one.then( () => console.log(1) );
const three = two.then( () => console.log(2) );
```

VS

```
Promise.resolve()
  .then( () => console.log(1) )
  .then( () => console.log(2) );
```

CATCH()

Promises `catch` method covers "failures"

- any thrown errors INSIDE a promise
- any returned **rejected** (vs **resolved** or **pending**) Promises

```
Promise.resolve()
  .then( () => {
    throw new Error("poop");
  })
  .then( () => console.log('does not happen') )
  .catch( err => console.log(err) );
```

`catch()` also returns a promise - **resolved** by default!

- Allows you to handle errors and keep going

TRY/CATCH IS USELESS WITH PROMISES!

```
try {
  Promise.resolve()
    .then( () => {
      console.log(1);
      throw new Error("poop");
    });
} catch(err) {
  // Doesn't happen
  console.log(`caught ${err}`);
}
console.log(2);
```

Why? (Hint: output is `2 1`)

ASYNC/AWAIT

A newer syntax is `async` and `await`

- A different way to manage promises
- Hides the `.then()` and `.catch()`
- Implicitly sets all following code to be async
- Allows try/catch

I do **not recommend** using async/await for this class

Until you know promises very comfortably, async/await can cause confusion by hiding what is really happening