ASYNCHRONOUS JAVASCRIPT

WEB PROGRAMMING

TODAY'S TOPICS

- The "function" expression
- Higher order functions
- The "arrow" function
- Event loop
  - The Callstack
  - WebAPIs
- Callback hell!
- Promises
- The "async" keyword
- The "await" keyword

THE “FUNCTION” EXPRESSION

FUNCTION EXPRESSION

- A function expression is very similar to and has almost the same syntax as a function declaration.

- The main difference between a function expression and a function declaration:
  - Function expressions are used to create *anonymous* functions (functions without function name).
  - Function expressions in JavaScript are not **hoisted**, unlike function declarations. You can't use function expressions before you create them.

LET'S SEE SOME CODE!

THE FUNCTION EXPRESSION

HIGHER ORDER FUNCTIONS

HIGHER ORDER FUNCTIONS

- Higher order functions are functions that accept other functions as arguments
  - Any function that is passed as an argument is called a "**callback**" function.
  - Functions that return a function

LET'S SEE SOME CODE!

HIGHER ORDER FUNCTIONS

THE ARROW FUNCTIONS

SYNTACTICALLY COMPACT ALTERNATIVE TO A REGULAR "FUNCTION" EXPRESSION

ARROW FUNCTIONS, THE BASICS

- There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.

- **Arrow functions** are handy for one-liners. They come in two flavors:
  - Without curly braces: `(...args) => expression` – the right side is an expression: the function evaluates it and returns the result.
  - With curly braces: `(...args) => { body }` – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.

LET’S SEE SOME CODE!

THE ARROW FUNCTIONS

setTimeout() & setInterval()

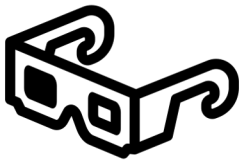
- The setTimeout() method calls a function or evaluates an expression after a specified number of milliseconds.

- The setInterval() method calls a function or evaluates an expression at specified intervals (in milliseconds).
  - The setInterval() method will continue calling the function until clearInterval() is called, or the window is closed.
  - The ID value returned by setInterval() is used as the parameter for the clearInterval() method.

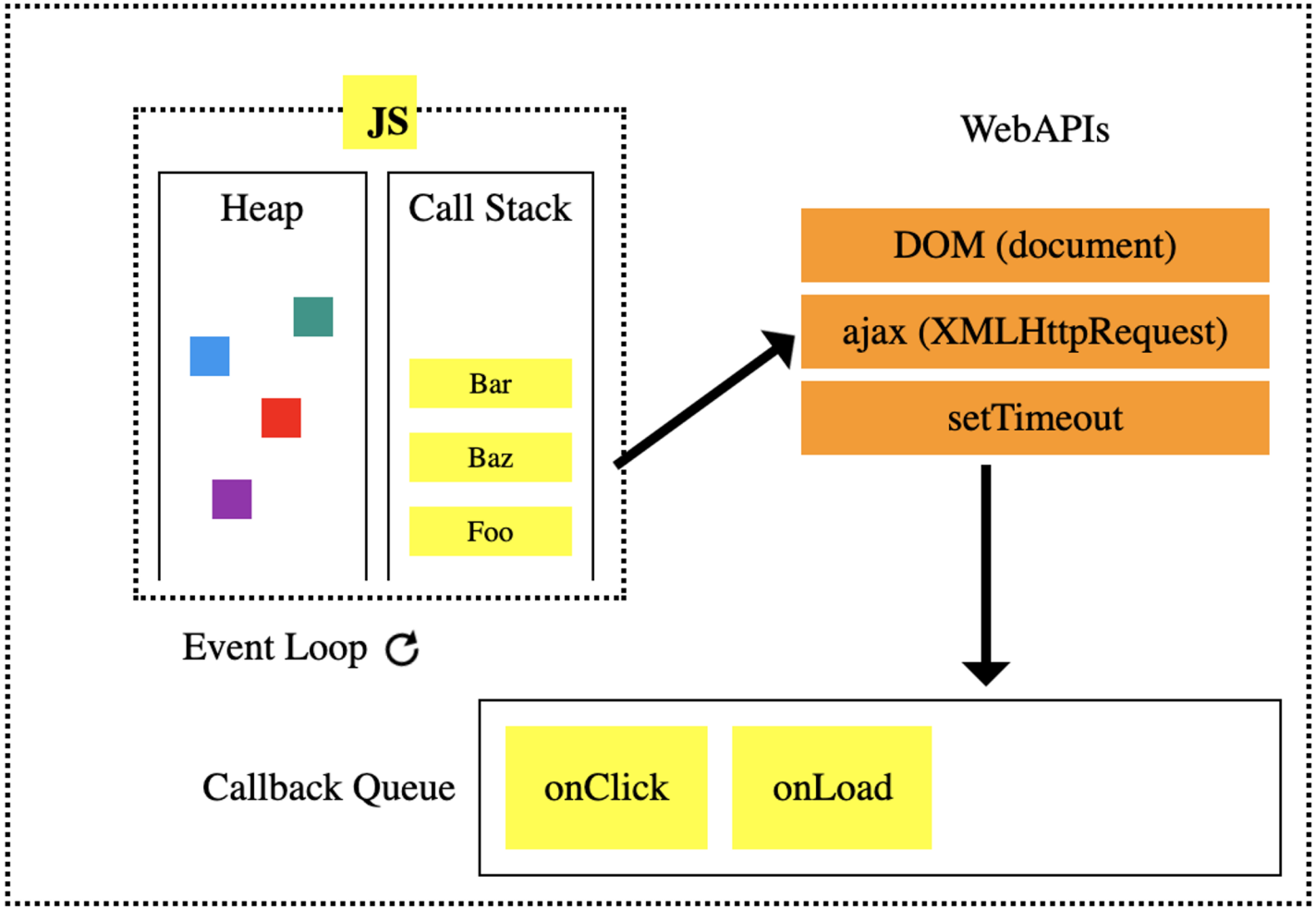
```
var myVar = setInterval(myTimer, 1000);

function myTimer() {
  var d = new Date();
  var t = d.toLocaleTimeString();
  document.getElementById("demo").innerHTML = t;
}

function myStopFunction() {
  clearInterval(myVar);
}
```

THE EVENT LOOP

Event Loop



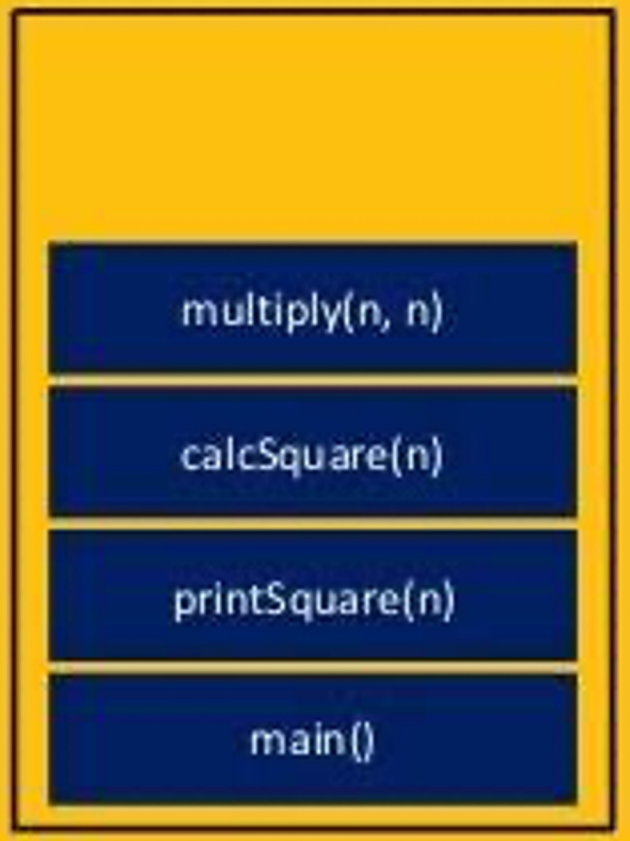
Let's see how it works



THE WEB APIS

- Browsers come with Web APIs (ref) that are able to handle certain tasks in the backgound
  - setTimeout()
  - Making AJAX requests
  - Manipulating DOM
- The JS call stack passes these tasks off to the browser to take care of.
- When the browser finishes those tasks, they return to "**Callback Queue**" and are later pushed to the **Call Stack** as a **Callback.**

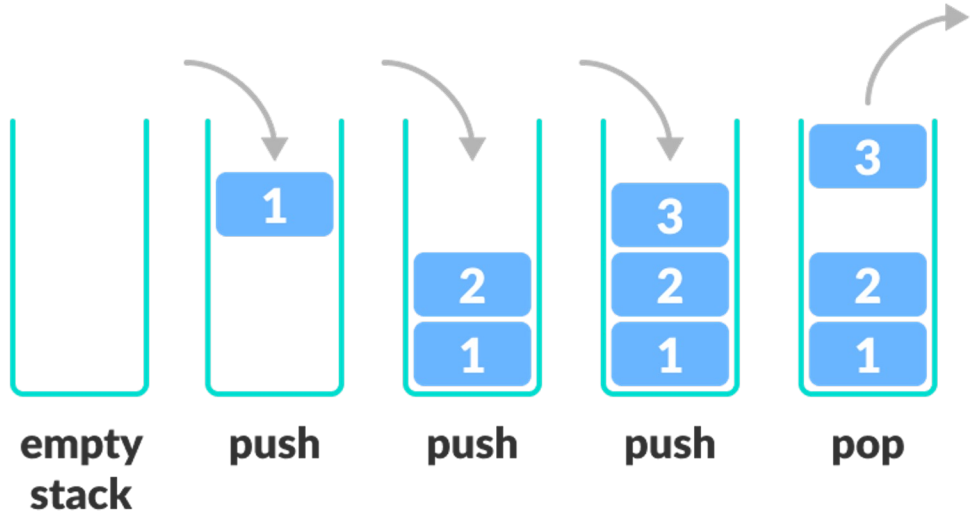
THE CALLSTACK

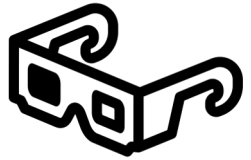


One thread == One call stack == One thing at a time

WHAT IS CALL STACK?

- The call stack is a Last-In, First-Out (LIFO) data structure containing the address at which execution will resume and often local variables and parameters from each call.
- Let's understand the call stack by a simple example.





Let's see how it works

CALLBACK HELL!!!

- **Seasoned JS developers must have heard tha term “Callback hell”. What does it mean?**

LET’S SEE SOME CODE!

PROMISES

WHAT IS A PROMISE?

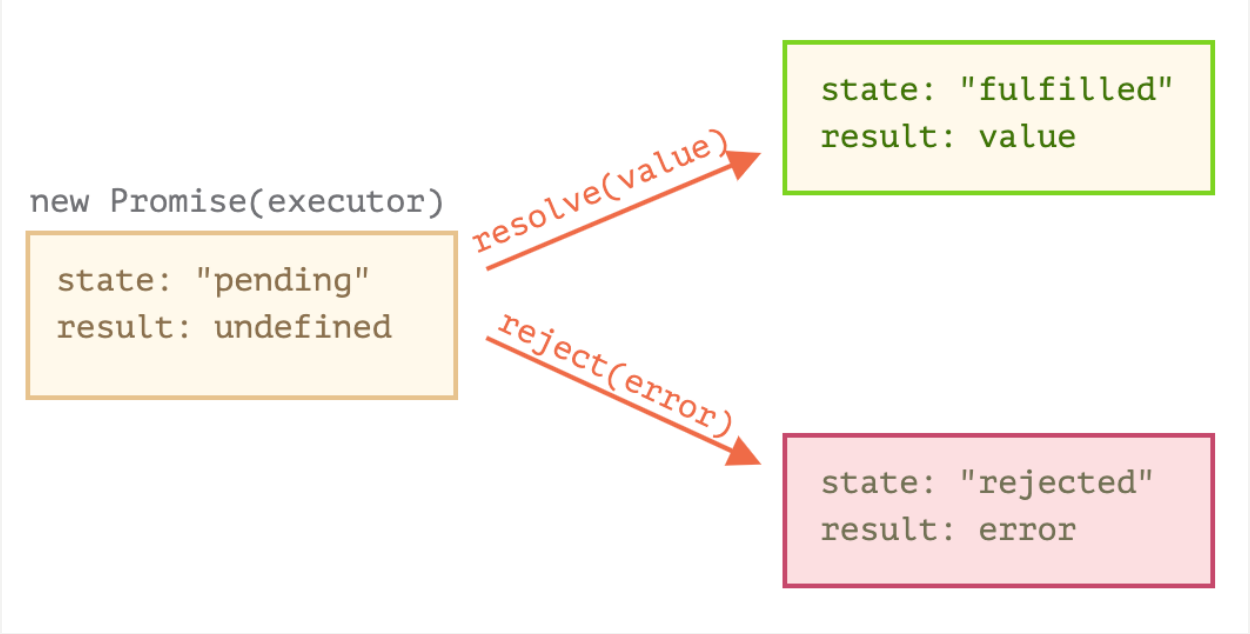
- A Promise in short:

"Imagine you are a **kid**. Your mom **promises** you that she'll get you a **new phone** next week."

- You *don't know* if you will get that phone until next week.
- Your mom can either *really buy* you a new phone, or *she doesn't*, because she is not happy :(

3 STATES OF PROMISE

- That is a **promise**. A promise has 3 states. They are:
  - **Pending**: You *don't know* if you will get that phone
  - **Fulfilled**: Mom is *happy*, she buys you a brand new phone
  - **Rejected**: Mom is *unhappy*, she doesn't buy you a phone




```
new Promise(function (resolve, reject) {  } );
```

Consumers: then, catch and finally

- promise.**then**()
  - The first argument of .**then** is a function that runs when the promise is resolved, and receives the result.
  - The second argument of .**then** is a function that runs when the promise is rejected, and receives the error.
- promise.**catch**()
  - The call .**catch**(func) is a complete analog of .**then**(null, func), it's just a shorthand.
- promise.**finally**()
  - .**finally**() is always run when the promise is settled: be it resolve or reject.

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

Let’s make some promises

ASYNC & AWAIT

THERE'S A SPECIAL SYNTAX TO WORK WITH PROMISES IN A MORE COMFORTABLE FASHION, CALLED "ASYNC/AWAIT".

ASYNC

- Let's start with the **async** keyword. It can be placed before a function, like this:

```
async function f() {
  return 1;
}
```

- The word "**async**" before a function means one simple thing: a function *always* returns a **promise**. Other values are wrapped in a resolved promise automatically.

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

AWAIT

```
// works only inside async functions
let value = await promise;
```

- The keyword **await** makes JavaScript wait until that promise settles and returns its result.
- Here's an example with a promise that resolves in 1 second:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

The function execution "pauses" at the line (*) and resumes when the promise settles, with result becoming its result.

So the code above shows "done!" in one second.

ERROR HANDLING

```
async function f() {
  throw new Error("Whoops!");
}
```

- If a promise resolves normally, then await promise returns the result. But in the case of a rejection, it throws the error, just as if there were a throw statement at that line.

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

- We can catch that error using **try..catch**, the same way as a regular throw:

SUMMARY

- The **async** keyword before a function has two effects:
  - Makes it always return a promise.
  - Allows await to be used in it.
- The **await** keyword before a promise makes JavaScript wait until that promise settles, and then:
  - If it's an error, the exception is generated — same as if throw error were called at that very place.
  - Otherwise, it returns the result.

LET'S SEE SOME CODE!

Async & Await