Lecture 4
Asynchronous JavaScript

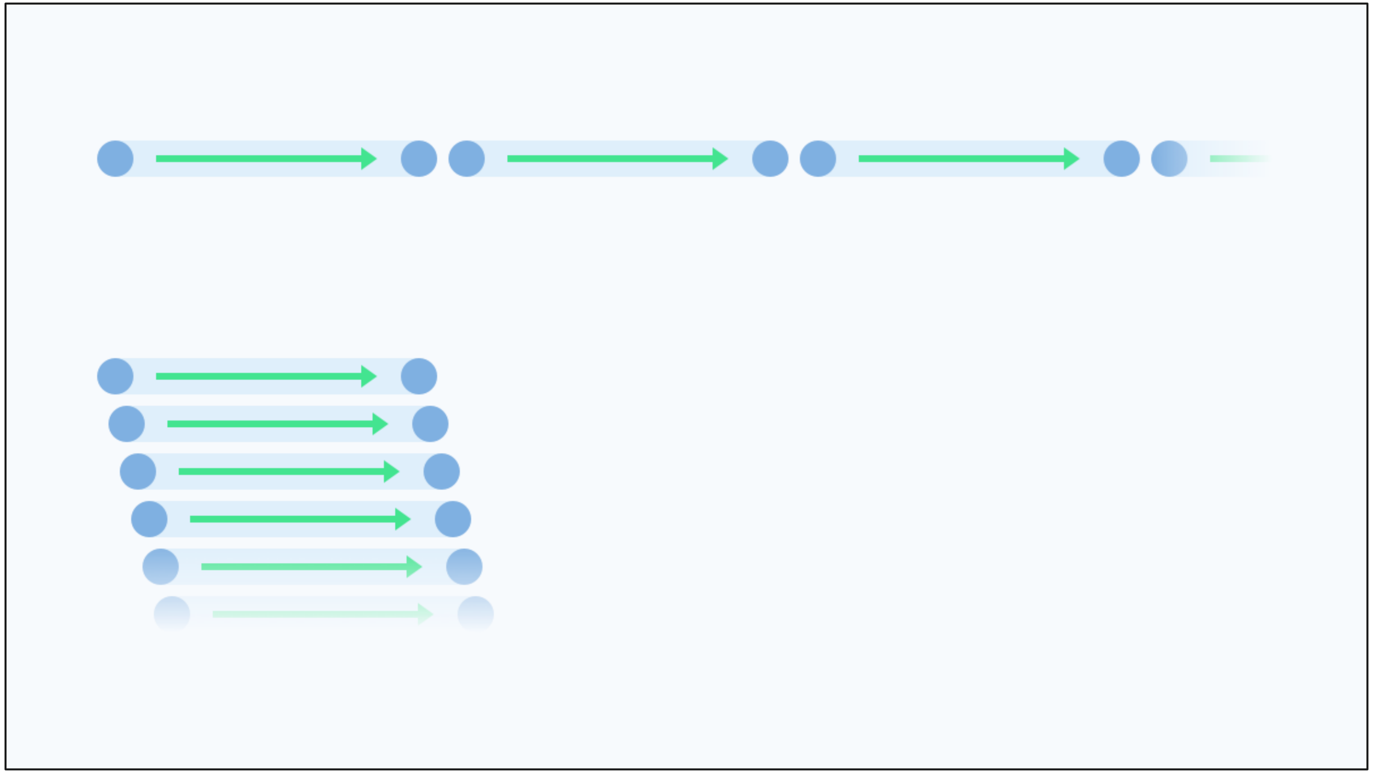
Today’s Contents

- Asynchronous and Promises
- Async/Await (with examples)
- JSON

Asynchronous Programming

- Callbacks
- Promises

Synchronous vs Asynchronous



Typical JS example

```
(function() {
  ...
  function init() {
    console.log('page loaded');
    qs('button').addEventListener('click', clickHandler);
    showMenu();
  }

  function showMenu() {
    id('menu').classList.remove('hidden');
  }

  function clickHandler() {
    /* Your code */
  }
})();
```

Callbacks?

Callbacks are a very powerful feature in event-driven programming.

It's useful to have the ability in the JavaScript language to pass callback functions as arguments to other functions like `addEventListener` and `setTimeout` in JS

Asynchronous Programming

The JS programs we've been writing are naturally asynchronous

We pass functions as arguments to other functions so that we can 'call back later' once we know something we expect occurred.

We’ve already been writing asynchronously!

```
btn.addEventListener('click', callbackFn);
btn.addEventListener('click', function() {
  ...
});
btn.addEventListener('click', () => {
  ...
});
```

```
setTimeout(callbackFn, 2000);
setTimeout(function() {
  ...
}, 2000);
setTimeout(() => {
  ...
}, 2000);
```

Why is JavaScript so different?

Java, and other compiled languages, are often used to build *systems*.

- Objects are great to compose together to build complex systems.
- Systems must be reliable - a benefit of strict types, compiling, and well-defined behavior in Java.

JavaScript is used to *interact* and *communicate*.

- It listens.
- It responds.
- It requests.

While programs in Java often have a well-defined specification (behavior), programs in JS has to deal with uncertainty (weird users, unavailable servers, no internet connection, etc.)

What if?

```
let myBtn = qs('button:nth-child(1)');
while (!myBtn.clicked) {
  // cross our fingers
}
console.log('Finally Been Clicked T_T');

let myBtn2 = qs('button:nth-child(2)');
while (!myBtn2.clicked) {
  // hold our breath
}
console.log('It was worth the wait...');
```

- This won't work (and will crash your browser)
- We *wouldn't be able to do anything* while we were waiting
- But the synchronous logic is nice
- What if we could could make our code feel more synchronous?

Analogy: You're out for pizza

At the restaurant you might follow these steps:

- Request menu
- Order pizza
- Check pizza
- Eat pizza
- Pay for pizza

Each step can't continue before the previous finishes.

What do you do in between?

```
requestMenu();
// twiddle thumbs
orderPizza();
// twiddle thumbs
verifyPizza();
// twiddle thumbs
eatPizza();
// twiddle thumbs
payForPizza();
```

Callback (again) to Callbacks:

We can imagine all of these steps as a series of callbacks, depending on the event previous to them:

```
function order() {
  setTimeout(function() {
    makeRequest('Requesting menu...');
    setTimeout(function() {
      makeRequest('Ordering pizza...');
      setTimeout(function() {
        makeRequest('Checking pizza...');
        setTimeout(function() {
          makeRequest('Eating pizza...');
          setTimeout(function() {
            makeRequest('Paying for pizza...');
            setTimeout(function() {
              let response = makeRequest('Done! Heading home.');
              console.log(response);
            }, ?);
          }, ?);
        }, ?);
      }, ?);
    }, ?);
  }, ?);
}
```

Callback (again) to Callbacks:

```
function order() {
  setTimeout(function() {
    makeRequest('Requesting menu...');
    setTimeout(function() {
      makeRequest('Ordering pizza...');
      setTimeout(function() {
        makeRequest('Checking pizza...');
        setTimeout(function() {
          makeRequest('Eating pizza...');
          setTimeout(function() {
            makeRequest('Paying for pizza...');
            setTimeout(function() {
              let response = makeRequest('Done! Heading home.');
              console.log(response);
            }, ?);
          }, ?);
        }, ?);
      }, ?);
    }, ?);
  }, ?);
}
```

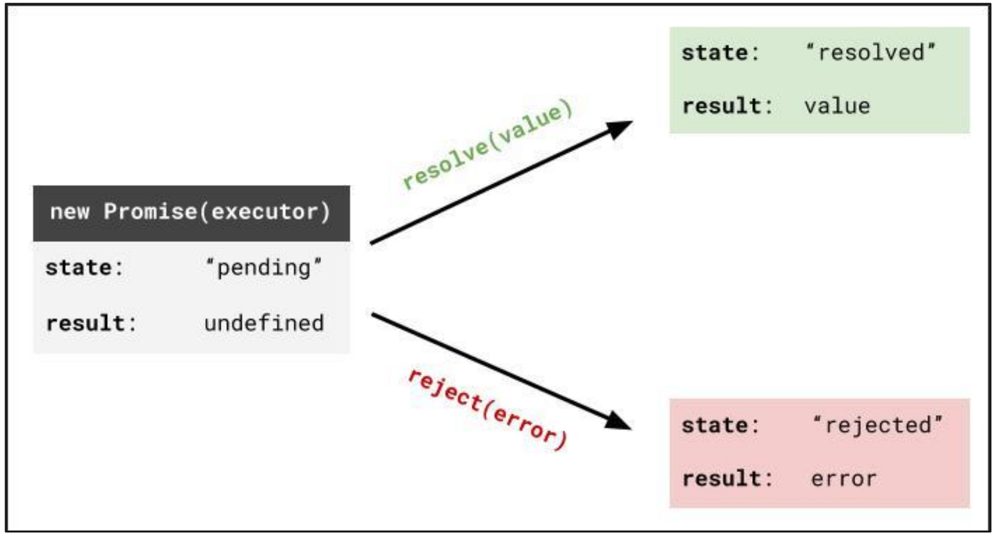
But how long should we wait before running each step?

Wouldn’t it be Nice...

... if we could do this?

```
orderPizza()
   .then(verify)
   .then(eat)
   .then(pay)
   .catch(badPizza);
```

Promises



Promises are a sort of contract:

- Something will happen
- You can have multiple things happen.
- And catch any errors.

Can only go from Pending to Fulfilled or Rejected (no takebacks)

Example: 'I promise to return to your table'

- Pending: Waiting for my pizza
- Fulfilled: Pizza has arrived!!
- Rejected: Kitchen ran out of cheese. :(

Promises on MDN

Creating a Promise

Function	Description
`let promiseObj = new Promise(executorFn)`	Creates a new Promise object with the executorFn
`promiseObj.then(onFulfilled, onRejected)`	Invokes the onFulfilled (onRejected) function when the promise is fulfilled (rejected)
`promiseObj.catch(callback)`	Invokes the callback function if the promise is rejected (or an error occurs)

```
function executorFn(resolve, reject) {
    // ...
    if (conditionMet) {
        resolve(); // Passed by the Promise object
    } else {
        reject(); // Passed by the Promise object
    }
}
```

You define this function and pass it into the Promise constructor

Back to that Pizza

```
function orderExecutor(resolve, reject) {
    // reject not used here
    console.log('making our pizza...');
    setTimeout(resolve, 5000);
}

let orderPizza = new Promise(orderExecutor);
orderPizza.then(function () {
    console.log('eating pizza!');
});
```

Back to that Pizza

We can pass a value to resolve...

```
function orderExecutor(resolve, reject) {
    console.log('making our pizza...');
    setTimeout(function() {
        resolve('Here\'s your pizza!');
    }, 5000);
}

let orderPizza = new Promise(orderExecutor);
orderPizza.then(function (value) {
    console.log(value);
});
```

That value gets passed to the function passed into `then`

Back to that Pizza

The functions passed to `then` can pass their returned values to the next `then` callback

```
function eat(value) {
    return value + ', and now it\'s gone';
}

let orderPizza = new Promise(orderExecutor);
orderPizza.then(eat).then(function (value) {
    console.log(value);
});
```

Back to that Pizza

You can also return other promises, which halt the execution of the next `then` callback until it's resolved

```
function eatExecutor(resolve, reject) {
    console.log('eating our pizza...');
    setTimeout(resolve, 3000);
}

function eat() {
    return new Promise(eatExecutor);
}

let orderPizza = new Promise(orderExecutor);
orderPizza.then(eat).then(function () {
    console.log('Paying the bill!');
});
```

Still Asynchronous

In what order do these log statements appear in the console?
Note that the `setTimeout` has been removed

```
function orderExecutor(resolve, reject) {
    console.log('Pizza ordered...');
    resolve('Here\'s your pizza!');
}

let orderPizza = new Promise(orderExecutor);
orderPizza.then(function (value) {
    console.log(value);
});
console.log('Waiting for my pizza!');
```

Still Asynchronous

In what order do these log statements appear in the console?

```
function orderExecutor(resolve, reject) {
    console.log('Pizza ordered...'); // 1
    resolve('Here\'s your pizza!');
}

let orderPizza = new Promise(orderExecutor);
orderPizza.then(function (value) {
    console.log(value); // 3
});
console.log('Waiting for my pizza!'); // 2
```

(*) Even if the Promise resolves immediately, any .then() chained onto it will be put into the microtask queue, whose tasks will only start running once all other *synchronous* code has finished.

Rejecting a Pizza

```
// MUST have both parameters defined
function orderExecutor(resolve, reject) {
    console.log('Pizza ordered...');
    setTimeout(function() {
        reject('Ran outta cheese. Can you believe it?');
    }, 2000);
}

let orderPizza = new Promise(orderExecutor);
orderPizza
    .then(function () { console.log('Woohoo, let's eat!'); })
    .catch(function (value) { console.log(value); });
```

Why are we using Promises?

- A `Promise` is used when we don't want to halt the main flow of execution but are dealing with a task that takes an uncertain amount of time
  - Example: requesting information from a server
    - What if the server is down? What if there is an error in what I get back? What if I made a request with inaccurate information?

Uncertainty

Some operations take an unknown amount of time or have a significant chance of failure

- File I/O
- Database transactions
- HTTP requests
    - Resource doesn't exist (404)
    - Really long response time or server is down
    - Bad internet connection

Whether these operations succeed or fail, we still want to do something in response

`then` and `catch` Return Promises

`then` and `catch` return *new* Promises

```
function executor(resolve) {
    resolve('Woohoo!');
}

let myPromise = new Promise(executor);
let thenPromise = myPromise.then(console.log);
let catchPromise = thenPromise.catch(console.error);

console.log(thenPromise instanceof Promise); // true
console.log(catchPromise instanceof Promise); // true

console.log(myPromise === thenPromise); // false
console.log(myPromise === catchPromise); // false

console.log(thenPromise === catchPromise); // false
```

`then` and `catch` Return Promises

```
function executor(resolve) {
    resolve('Woohoo!');
}

function processStr(val) {
    // mellow out that message a bit
    return val.toLowerCase().replace('!', '');
}

let myPromise = new Promise(executor);
let thenPromise = myPromise.then(processStr);
console.log(thenPromise);
```

`processStr` returns a string, but `then` turns it into a Promise that immediately resolves with the value `"woohoo"`

`then` and `catch` Return Promises

```
function executor(resolve) {
    resolve('Woohoo!');
}

function processStr(val) {
    // mellow out that message a bit
    return new Promise(function(resolve) {
        resolve(val.toLowerCase().replace('!', ''));
    });
}

let myPromise = new Promise(executor);
let thenPromise = myPromise.then(processStr);
console.log(thenPromise);
```

The code in this slide is equivalent to the previous

`then` and `catch` Return Promises

```
function executor(resolve) {
    resolve('Woohoo!');
}

function processStr(val) {
    // mellow out that message a bit
    return new Promise(function(resolve) {
        setTimeout(function() {
            resolve(val.toLowerCase().replace('!', ''));
        }, 5000);
    });
}

let myPromise = new Promise(executor);
let thenPromise = myPromise.then(processStr);
console.log(thenPromise);
```

Now, `thenPromise` is "PENDING" and won't resolve until the promise returned by `processStr` resolves.

Promises to the Rescue

This chaining of promises is what makes the below possible

```
orderPizza()
   .then(eat)
   .then(pay)
   .catch(badPizza);
```

Async/Await

- Using async/await
- Catching errors
- Async functions

What if?

Now we're back to this example:

```
let myBtn = qs('button:nth-child(1)');
while (!myBtn.clicked) {
  // twiddle our thumbs
}
console.log('Finally Been Clicked');

let myBtn2 = qs('button:nth-child(2)');
while (!myBtn2.clicked) {
  // twiddle our thumbs
}
console.log('Click 2');
```

What if? (with Promises)

```
function firstBtnClick() {
    return new Promise(function (resolve) {
        let myBtn = qs('button:nth-child(1)');
        myBtn.addEventListener('click', resolve);
    });
}

function nextBtnClick() {
    return new Promise(function (resolve) {
        let myBtn = qs('button:nth-child(2)');
        myBtn.addEventListener('click', resolve);
    });
}

firstBtnClick()
    .then(() => { console.log('Finally Been Clicked'); })
    .then(nextBtnClick)
    .then(() => { console.log('Click 2'); });
```

What if? (with Promises + A Little Syntactic Sugar)

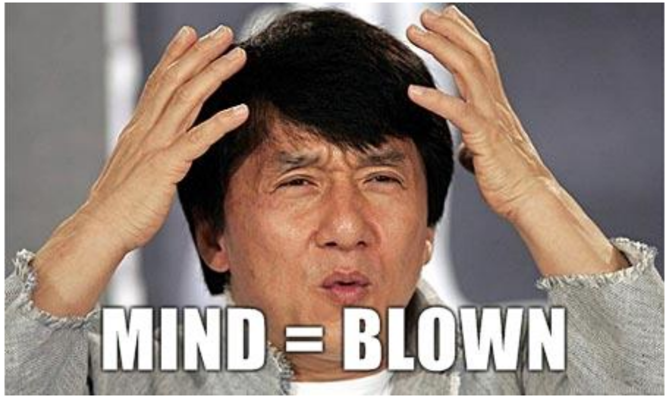
```
function firstBtnClick() {
    return new Promise(function (resolve) {
        let myBtn = qs('button:nth-child(1)');
        myBtn.addEventListener('click', resolve);
    });
}

function nextBtnClick() {
    return new Promise(function (resolve) {
        let myBtn = qs('button:nth-child(2)');
        myBtn.addEventListener('click', resolve);
    });
}

await firstBtnClick();
console.log('Finally Been Clicked');
await nextBtnClick();
console.log('Click 2');
```

Mind = Blown

```
await firstBtnClick();
console.log('Finally Been Clicked');
await nextBtnClick();
console.log('Click 2');
```



Async/Await

"Syntactic sugar" that wraps a function's return in a promise
Allows code to "wait" for the thing to return.

```
async function sayHelloAsync(name) {
  return "Hello " + name;
}

console.log(sayHelloAsync("dubs")); // Promise <pending>
let message = await sayHelloAsync("dubs");
console.log(message); // "Hello dubs"
```

- `async` does the same thing to functions that `then` does
    - It wraps the return value in a Promise whose resolved value is the return value
- `await` halts execution of the code until the Promise is resolved and then returns the resolved value of the promise

Async/Await

```
async function sayHelloAsync(name) {
  return "Hello " + name;
}
```

is the same as

```
function sayHelloAsync(name) {
   return new Promise(function(resolve) {
      resolve("Hello " + name);
   });
}
```

Back to that Pizza

```
function orderExecutor(resolve, reject) { // reject not required here
    console.log('making our pizza...');
    setTimeout(resolve, 5000);
}

await new Promise(orderExecutor);
console.log('eating pizza!');
```

Back to that Pizza

We can pass a value to resolve...

```
function orderExecutor(resolve, reject) {
    console.log('making our pizza...');
    setTimeout(function() {
        resolve("Here's your pizza!");
    }, 5000);
}

let pizza = await new Promise(orderExecutor);
console.log(pizza);
```

That value will be the returned value of the awaited statement

Back to that Pizza

```
async function eat(value) {
    return value + ", and now it's gone";
}

let pizza = await new Promise(orderExecutor);
let eatingResult = await eat(pizza);
console.log(eatingResult);
```

Back to that Pizza

You can also return other promises, which halt the execution of the next `then` callback until it's resolved

```
function eatExecutor(resolve, reject) {
    console.log('eating our pizza...');
    setTimeout(resolve, 3000);
}

async function eat() { // don't need async here... why?
    return new Promise(eatExecutor);
}

let pizza = await new Promise(orderExecutor);
let eatingResult = await eat();
console.log('Paying the bill!');
```

Still Asynchronous... or is it?

In what order do these log statements appear in the console?

```
function orderExecutor(resolve, reject) {
    console.log('Pizza ordered...');
    setTimeout(function() {
        resolve("Here's your pizza!");
    }, 3000);
}

let pizza = await new Promise(orderExecutor);
console.log(pizza);
console.log('Waiting around');
```

Rejecting a Pizza

```
// MUST have both parameters passed in
function orderExecutor(resolve, reject) {
    console.log('Pizza ordered...');
    setTimeout(function() {
        reject("Ran outta cheeese. Can you believe it?");
    }, 2000);
}

try {
    let pizza = await new Promise(orderExecutor);
    console.log("Woohoo, let's eat!");
} catch (error) {
    console.log(error);
}
```

Error handling with `async/await`

For error-handling with `async/await`, you must use `try/catch` instead of `.then/.catch`

The `catch` statement will catch any errors that occur in the then block (whether it’s in a Promise or a syntax error in the function), similar to the `.catch` in a promise chain

When Do I Need the Keyword `async`?

For any function that is `await`'d, but that doesn't return a promise (although it'll still work to add `async` even if it does)

```
async function eat(value) {
    return value + ", and now it's gone";
}
let pizza = await new Promise(orderExecutor);
let eatingResult = await eat(pizza); // don't need to do this.
Why not?
console.log(eatingResult);
```

For any function that that uses `await` in its implementation

```
async function orderPizza() {
    let pizza = await new Promise(orderExecutor);
    return 'Done!';
}
console.log(await orderPizza());
console.log('What now?');
```

Question: Why are `async/await` helpful?

- Makes our code "look" synchronous again (why is *this* helpful?)
- Don't need callbacks, can just have regular functions
- Can use data outside of a `.then` chain.

Should I use the `.then`/`.catch` chain or `async`/`await`?

- It doesn’t matter, choose what you prefer and be consistent
  - Unless explicitly specified in assignments (rarely happens)
- There will never be a situation in which you can’t use one method over the other
- However, you should know how to use both

Data? → JavaScript Objects

In JavaScript, you can create a new object without creating a "class" like you do in Java

```
let myobj = {
  fieldName1: value1,
  ...
  fieldName: value
};
```

```
let bestCourse = {
  dept : "FIT",
  code : 62,
  qtr : "fall2023",
  sections : ["AB", "AC", "AD", "AE"]
};
```

You can add properties to any object even after it is created:

```
bestCourse.mascot = "pokemon";
```

Example of JS Object

An object can have methods (function properties) that refer to itself as this, we can refer to the fields with `.fieldName` or `["fieldName"]` syntax. Find out more info here.

```
let data = {
  "name": "FIT",
  "course-num": 101,
  "hello" : function() {
    console.log("welcome to FIT101!");
  },
  "age" : "i am not going to tell y'all how old i am",
  "favorites": ["survivor", "podcasts", "peanut butter"]
};
console.log(data.favorites[1]);  // podcasts
data.hello();                    // welcome to FIT101!
console.log(data["course-num"]); // 101
console.log(data.name);         // FIT
```

Examples of JS objects we’ve seen so far

- DOM elements
- `document`, `window`

JavaScript Objects vs. JSON

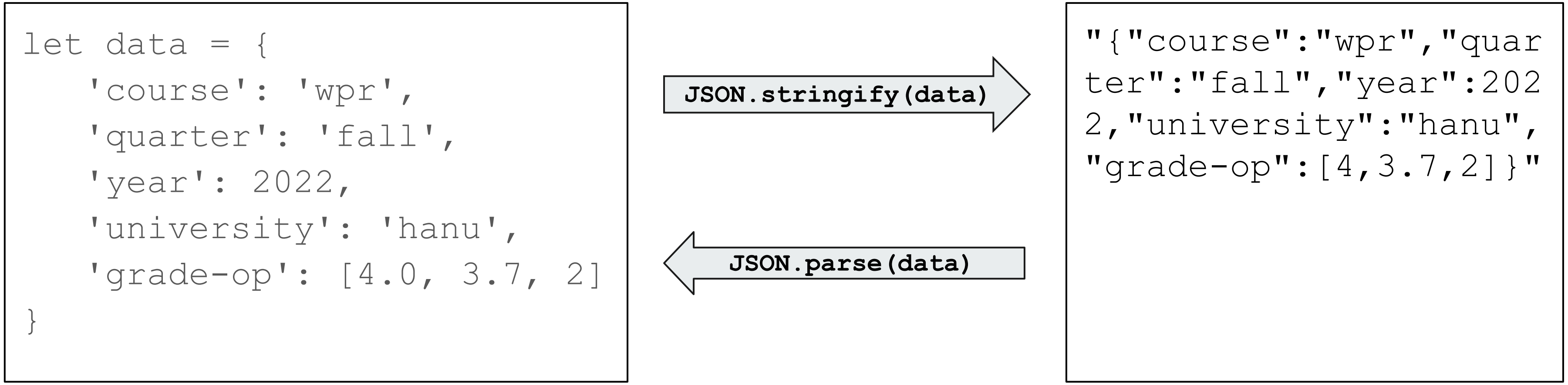
JSON is a way of representing objects, or structured data.

- (The technical term is "serializing" which is just a fancy word for turning an object into a savable string of characters)

Browser JSON methods:

- `JSON.parse( /* JSON string */ )` -- converts JSON string into Javascript object
  (Deserialization)
- `JSON.stringify( /* Javascript Object */ )` -- converts a Javascript object into JSON text
  (Serialization)

Browser JSON methods



```
let data = {
    'course': 'wpr',
    'quarter': 'fall',
    'year': 2022,
    'university': 'hanu',
    'grade-op': [4.0, 3.7, 2]
}
```

```
"{"course":"wpr","quar
ter":"fall","year":202
2,"university":"hanu",
"grade-op":[4,3.7,2]}"
```

JSON Limitations

JSON can't handle certain data types, so these things just fall out of the object if you try to make JSON strings out of them:

- Function
- Date
- RegExp
- Error

Since JSON is ideal for communicating across different types of systems, you can't put Javascript functions in JSON. Other languages wouldn't be able to read JSON effectively if it had Javascript code in it.

(This is also why Dates and RegExps can't go into the JSON object -- other languages wouldn't know how to interpret them for what they are.)

There are a few other JSON rules which you can get more details in the reading. Numerous validators/formatters available, e.g. JSONLint